

# AWS 云计算实战

## AWS IN ACTION

[德] 安德烈亚斯·威蒂格 (Andreas Wittig) 著  
迈克尔·威蒂格 (Michael Wittig) 译  
费良宏 张波 黄涛



# 目录

[版权信息](#)

[版权声明](#)

[内容提要](#)

[译者序](#)

[序](#)

[前言](#)

[资源与支持](#)

[致谢](#)

[关于本书](#)

[关于作者](#)

[关于封面插画](#)

[第一部分 AWS云计算起步](#)

[第1章 什么是Amazon Web Services](#)

[1.1 什么是云计算](#)

[1.2 AWS可以做什么](#)

[1.2.1 托管一家网店](#)

[1.2.2 在专有网络内运行一个Java EE应用](#)

[1.2.3 满足法律和业务数据归档的需求](#)

[1.2.4 实现容错的系统架构](#)

[1.3 如何从使用AWS上获益](#)

[1.3.1 创新和快速发展的平台](#)

- [1.3.2 解决常见问题的服务](#)
- [1.3.3 启用自动化](#)
- [1.3.4 灵活的容量（可扩展性）](#)
- [1.3.5 为失效而构建（可靠性）](#)
- [1.3.6 缩短上市的时间](#)
- [1.3.7 从规模经济中受益](#)
- [1.3.8 全球化](#)
- [1.3.9 专业的合作伙伴](#)
- [1.4 费用是多少](#)
- [1.4.1 免费套餐](#)
- [1.4.2 账单样例](#)
- [1.4.3 按使用付费的机遇](#)
- [1.5 同类对比](#)
- [1.6 探索AWS服务](#)
- [1.7 与AWS交互](#)
- [1.7.1 管理控制台](#)
- [1.7.2 命令行接口](#)
- [1.7.3 SDK](#)
- [1.7.4 蓝图](#)
- [1.8 创建一个AWS账户](#)
- [1.8.1 注册](#)
- [1.8.2 登录](#)
- [1.8.3 创建一个密钥对](#)
- [1.8.4 创建计费告警](#)
- [1.9 小结](#)

## [第2章 一个简单示例：5分钟搭建WordPress站点](#)

- [2.1 创建基础设施](#)
- [2.2 探索基础设施](#)
- [2.2.1 资源组](#)
- [2.2.2 Web服务器](#)
- [2.2.3 负载均衡器](#)
- [2.2.4 MySQL数据库](#)
- [2.3 成本是多少](#)
- [2.4 删除基础设施](#)
- [2.5 小结](#)

## [第二部分 搭建包含服务器和网络的虚拟基础设施](#)

### [第3章 使用虚拟服务器：EC2](#)

#### [3.1 探索虚拟服务器](#)

##### [3.1.1 启动虚拟服务器](#)

##### [3.1.2 连接到虚拟服务器](#)

##### [3.1.3 手动安装和运行软件](#)

#### [3.2 监控和调试虚拟服务器](#)

##### [3.2.1 显示虚拟服务器的日志](#)

##### [3.2.2 监控虚拟服务器的负载](#)

#### [3.3 关闭虚拟服务器](#)

#### [3.4 更改虚拟服务器的容量](#)

#### [3.5 在另一个数据中心开启虚拟服务器](#)

#### [3.6 分配一个公有IP地址](#)

#### [3.7 向虚拟服务器添加额外的网络接口](#)

#### [3.8 优化虚拟服务器的开销](#)

##### [3.8.1 预留虚拟服务器](#)

##### [3.8.2 对未使用的虚拟服务器竞价](#)

#### [3.9 小结](#)

### [第4章 编写基础架构：命令行、SDK和CloudFormation](#)

#### [4.1 基础架构即代码](#)

##### [4.1.1 自动化和DevOps运作](#)

##### [4.1.2 开发一种基础架构语言：JIML](#)

#### [4.2 使用命令行接口](#)

##### [4.2.1 安装CLI](#)

##### [4.2.2 配置CLI](#)

##### [4.2.3 使用CLI](#)

#### [4.3 使用SDK编程](#)

##### [4.3.1 使用SDK控制虚拟服务器：nodecc](#)

##### [4.3.2 nodecc如何创建一台服务器](#)

##### [4.3.3 nodecc是如何列出服务器并显示服务器的详细信息](#)

##### [4.3.4 nodecc如何终止一台服务器](#)

#### [4.4 使用蓝图来启动一台虚拟服务器](#)

##### [4.4.1 CloudFormation模板解析](#)

##### [4.4.2 创建第一个模板](#)

#### [4.5 小结](#)



## [第5章 自动化部署：CloudFormation、Elastic Beanstalk和OpsWorks](#)

### [5.1 在灵活的云环境中部署应用程序](#)

### [5.2 使用CloudFormation在服务器启动时运行脚本](#)

#### [5.2.1 在服务器启动时使用用户数据来运行脚本](#)

#### [5.2.2 在虚拟服务器上部署OpenSwan作为VPN服务器](#)

#### [5.2.3 从零开始，而不是更新已有的服务器](#)

### [5.3 使用Elastic Beanstalk部署一个简单的网站应用](#)

#### [5.3.1 Elastic Beanstalk的组成部分](#)

#### [5.3.2 使用Elastic Beanstalk部署一个Node.js应用Etherpad](#)

### [5.4 使用OpsWorks部署多层架构应用](#)

#### [5.4.1 OpsWorks的组成部分](#)

#### [5.4.2 使用OpsWorks部署一个IRC聊天应用](#)

### [5.5 比较部署工具](#)

#### [5.5.1 对部署工具分类](#)

#### [5.5.2 比较部署服务](#)

### [5.6 小结](#)

## [第6章 保护系统安全：IAM、安全组和VPC](#)

### [6.1 谁该对安全负责](#)

### [6.2 使软件保持最新](#)

#### [6.2.1 检查安全更新](#)

#### [6.2.2 在服务器启动时安装安全更新](#)

#### [6.2.3 在服务器运行时安装安全更新](#)

### [6.3 保护AWS账户安全](#)

#### [6.3.1 保护AWS账户的root用户安全](#)

#### [6.3.2 IAM服务](#)

#### [6.3.3 用于授权的策略](#)

#### [6.3.4 用于身份认证的用户和用于组织用户的组](#)

#### [6.3.5 用于认证AWS的角色](#)

### [6.4 控制进出虚拟服务器的网络流量](#)

#### [6.4.1 使用安全组控制虚拟服务器的流量](#)

#### [6.4.2 允许ICMP流量](#)

#### [6.4.3 允许SSH流量](#)

#### [6.4.4 允许来自源IP地址的SSH流量](#)

#### [6.4.5 允许来自源安全组的SSH流量](#)

#### [6.4.6 用PuTTY进行代理转发](#)

### [6.5 在云中创建一个私有网络：虚拟私有云](#)

- [6.5.1 创建VPC和IGW](#)
- [6.5.2 定义公有堡垒主机子网](#)
- [6.5.3 添加私有Apache网站服务器子网](#)
- [6.5.4 在子网中启动服务器](#)
- [6.5.5 通过NAT服务器从私有子网访问互联网](#)
- [6.6 小结](#)

## [第三部分 在云上保存数据](#)

### [第7章 存储对象：S3和Glacier](#)

- [7.1 对象存储的概念](#)
- [7.2 Amazon S3](#)
- [7.3 备份用户的数据](#)
- [7.4 归档对象以优化成本](#)
  - [7.4.1 创建S3存储桶配合Glacier使用](#)
  - [7.4.2 添加生命周期规则到存储桶](#)
  - [7.4.3 测试Glacier和生命周期规则](#)
- [7.5 程序的方式存储对象](#)
  - [7.5.1 设置S3存储桶](#)
  - [7.5.2 安装使用S3的互联网应用](#)
  - [7.5.3 检查使用SDK访问S3的代码](#)
- [7.6 使用S3来实现静态网站托管](#)
  - [7.6.1 创建存储桶并上传一个静态网站](#)
  - [7.6.2 配置存储桶来实现静态网站托管](#)
  - [7.6.3 访问S3上托管的静态网站](#)
- [7.7 对象存储的内部机制](#)
  - [7.7.1 确保数据一致性](#)
  - [7.7.2 选择合适的键](#)
- [7.8 小结](#)

### [第8章 在硬盘上存储数据：EBS和实例存储](#)

- [8.1 网络附加存储](#)
  - [8.1.1 创建EBS卷并挂载到服务器](#)
  - [8.1.2 使用弹性数据块存储](#)
  - [8.1.3 玩转性能](#)
  - [8.1.4 备份数据](#)
- [8.2 实例存储](#)

- [8.2.1 使用实例存储](#)
- [8.2.2 性能测试](#)
- [8.2.3 备份数据](#)
- [8.3 比较块存储解决方案](#)
- [8.4 使用实例存储和EBS卷提供共享文件系统](#)
- [8.4.1 NFS的安全组](#)
- [8.4.2 NFS服务器和卷](#)
- [8.4.3 NFS服务器安装和配置脚本](#)
- [8.4.4 NFS客户端](#)
- [8.4.5 通过NFS共享文件](#)
- [8.5 小结](#)

## [第9章 使用关系数据库服务：RDS](#)

- [9.1 启动一个MySQL数据库](#)
  - [9.1.1 用Amazon RDS数据库启动WordPress平台](#)
  - [9.1.2 探索使用MySQL引擎的RDS数据库实例](#)
  - [9.1.3 Amazon RDS的定价](#)
- [9.2 将数据导入数据库](#)
- [9.3 备份和恢复数据库](#)
  - [9.3.1 配置自动快照](#)
  - [9.3.2 手动创建快照](#)
  - [9.3.3 恢复数据库](#)
  - [9.3.4 复制数据库到其他的区域](#)
  - [9.3.5 计算快照的成本](#)
- [9.4 控制对数据库的访问](#)
  - [9.4.1 控制对RDS数据库的配置的访问控制](#)
  - [9.4.2 控制对RDS数据库的网络访问](#)
  - [9.4.3 控制数据访问](#)
- [9.5 可以依赖的高可用的数据库](#)
- [激活RDS数据库的高可用部署选项](#)
- [9.6 调整数据库的性能](#)
  - [9.6.1 增加数据库资源](#)
  - [9.6.2 使用读副本来增加读性能](#)
- [9.7 监控数据库](#)
- [9.8 小结](#)

## [第10章 面向NoSQL数据库服务的编程：DynamoDB](#)

- [10.1 操作DynamoDB](#)
  - [10.1.1 管理](#)
  - [10.1.2 价格](#)
  - [10.1.3 与RDS对比](#)
- [10.2 开发者需要了解的DynamoDB内容](#)
  - [10.2.1 表、项目和属性](#)
  - [10.2.2 主键](#)
  - [10.2.3 与其他NoSQL数据库的对比](#)
  - [10.2.4 DynamoDB本地版](#)
- [10.3 编写任务管理应用程序](#)
- [10.4 创建表](#)
  - [10.4.1 使用分区键的用户表](#)
  - [10.4.2 使用分区键和排序键的任务表](#)
- [10.5 添加数据](#)
  - [10.5.1 添加一个用户](#)
  - [10.5.2 添加一个任务](#)
- [10.6 获取数据](#)
  - [10.6.1 提供键来获取数据](#)
  - [10.6.2 使用键和过滤来查询](#)
  - [10.6.3 更灵活地使用二级索引查询数据](#)
  - [10.6.4 扫描和过滤表数据](#)
  - [10.6.5 最终一致地数据提取](#)
- [10.7 删除数据](#)
- [10.8 修改数据](#)
- [10.9 扩展容量](#)
- [10.10 小结](#)

## [第四部分 在AWS上搭架构](#)

- [第11章 实现高可用性：可用区、自动扩展以及CloudWatch](#)
  - [11.1 使用CloudWatch恢复失效的服务器](#)
    - [11.1.1 建立一个CloudWatch告警](#)
    - [11.1.2 基于CloudWatch对虚拟服务器监控与恢复](#)
  - [11.2 从数据中心故障中恢复](#)
    - [11.2.1 可用区：每个区域有多个数据中心](#)
    - [11.2.2 使用自动扩展确保虚拟服务器一直运行](#)
    - [11.2.3 在另一个可用区中通过自动扩展恢复失效的虚拟服务器](#)

- [11.2.4 陷阱：网络附加存储恢复](#)
- [11.2.5 陷阱：网络接口恢复](#)
- [11.3 分析灾难恢复的需求](#)
- [单个虚拟服务器的RTO和RPO的比较](#)
- [11.4 小结](#)

## [第12章 基础设施解耦：ELB与SQS](#)

- [12.1 利用负载均衡器实现同步解耦](#)
  - [12.1.1 使用虚拟服务器设置负载均衡器](#)
  - [12.1.2 陷阱：过早地连接到服务器](#)
  - [12.1.3 更多使用场景](#)
- [12.2 利用消息队列实现异步解耦](#)
  - [12.2.1 将同步过程转换成异步过程](#)
  - [12.2.2 URL2PNG应用的架构](#)
  - [12.2.3 创建消息队列](#)
  - [12.2.4 以程序化的方法处理消息](#)
  - [12.2.5 程序化地消费消息](#)
  - [12.2.6 SQS消息传递的局限性](#)
- [12.3 小结](#)

## [第13章 容错设计](#)

- [13.1 使用冗余EC2实例提高可用性](#)
  - [13.1.1 冗余可以去除单点故障](#)
  - [13.1.2 冗余需要解耦](#)
- [13.2 使代码容错的注意事项](#)
  - [13.2.1 让其崩溃，但也重试](#)
  - [13.2.2 幂等重试使得容错成为可能](#)
- [13.3 构建容错Web应用：Imagery](#)
  - [13.3.1 幂等图片状态机](#)
  - [13.3.2 实现容错Web服务](#)
  - [13.3.3 实现容错的工作进程来消费SQS消息](#)
  - [13.3.4 部署应用](#)
- [13.4 小结](#)

## [第14章 向上或向下扩展：自动扩展和CloudWatch](#)

- [14.1 管理动态服务器池](#)
- [14.2 使用监控指标和时间计划触发扩展](#)



[14.2.1 基于时间计划的扩展](#)

[14.2.2 基于CloudWatch参数的扩展](#)

[14.3 解耦动态服务器池](#)

[14.3.1 由负载均衡器同步解耦扩展动态服务器池](#)

[14.3.2 队列异步解耦扩展动态服务器池](#)

[14.4 小结](#)

## 版权信息

书名：AWS云计算实战

ISBN：978-7-115-48486-4

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

---

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

---

著 [德] 安德烈亚斯·威蒂格（Andreas Wittig）

迈克尔·威蒂格（Michael Wittig）

译 费良宏 张 波 黄 涛

责任编辑 杨海玲

---

人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

---

读者服务热线：(010)81055410

反盗版热线： (010)81055315

## 版权声明

Original English language edition, entitled *Amazon Web Services in Action* by Andreas Wittig and Michael Wittig published by Manning Publications Co., 209 Bruce Park Avenue, Greenwich, CT 06830. Copyright © 2016 by Manning Publications Co.

Simplified Chinese-language edition copyright © 2018 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由**Manning Publications Co.**授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

## 内容提要

Amazon Web Services (AWS) 是亚马逊公司的云计算平台，它提供了一整套基础设施和应用程序服务，可以帮助用户在云中运行几乎一切应用程序。本书介绍了AWS云平台的核心服务，如计算、存储和网络等内容。读者还可以从本书中了解在云上实现自动化、保证安全、实现高可用和海量扩展的系统架构的最佳实践。

本书分4个部分，共14章。本书从介绍AWS的基本概念开始，引入具体的应用示例，让读者对云计算和AWS平台有一个整体的了解；然后讲解如何搭建包含服务器和网络的基础设施；在此基础上，深入介绍如何在云上存取数据，让读者熟悉存储数据的方法和技术；最后展开讨论在AWS上如何设计架构，了解实现高可用性、高容错率和高扩展性的最佳实践。

本书适合对AWS感兴趣的运维人员和开发人员，尤其是那些需要将分布式应用向AWS平台迁移的运维人员和开发人员阅读。



## 译者序

大约30年前，我第一次见到了传说中的计算机——名为IBM PC的个人电脑。以今天的眼光来看，这台设备体型庞大、磁盘吱吱作响、操作界面极其简单，但是在那个时代能够操作这台设备的人简直是凤毛麟角。在其他同学面前，我不由得暗自得意，未来是计算机的时代，而我们注定是这个时代的佼佼者。

大约25年前，我第一次通过一台终端连接上了互联网，那个年代极少有人知道的新玩意儿。尽管当时互联网上的资源还是少得可怜，可这不妨碍我用Gopher、FTP还有Mosaic玩得不亦乐乎。我相信我接触到的是一个远未开发出来的金矿，它一定会改变我们的生活。

大约10年前，通过IT媒体我听到了那个时间最热门的概念——云计算。尽管关于什么是云计算以及云计算究竟会产生怎样的影响还有很大的争议，但是基于互联网提供计算服务的这个想法还是让我感到无比的振奋。我知道延续技术发展的脉络，未来的计算机以及互联网注定要融入云计算这个外延无限广阔的概念当中。

时至今日，我们生活中用到的手机打车、移动支付、社交媒体、新闻娱乐还有各种各样的新奇的应用无一例外都是依托于云计算而风起云涌般地出现的。而诸如联合利华、GE、飞利浦以及时代集团这样的传统企业也在利用云计算实现其数字化转型。无疑，云计算已经是这个时代的“新常态”。

今天当我们谈论起云计算，总会让我联想到30年前的个人电脑以及25年前互联网的出现。我相信云计算带来的改变和冲击不会亚于前两次技术进步的浪潮，而我们注定要在这个变革时代做出选择：成为赢家或者被时代淘汰。

我们选择翻译这一本书的目的，就是想让更多的读者了解到当前云计算发展的状态。通过实践让自己成为弄潮儿。在我的职业经历中，已经见识过太多的因为不适应技术进步而被淘汰的输家。当这一次变革到来的时刻，我们希望每一位读者都可以是时代的赢家。

感谢人民邮电出版社编辑对我们的支持，没有他们的努力这本书不会面世。在此一并致谢。

# 序

在整个20世纪90年代末和21世纪初，作为系统管理员，我负责维护网络服务在线、安全和确保用户的正常使用。当时，维护系统是一件乏味、单调的事情，涉及网络电缆吊装、服务器机架、从光盘介质安装操作系统和手动配置软件。任何希望从事新兴在线网络市场的企业都要承担管理物理服务器、接受相关资金成本和运营成本的压力，并希望获得足够的成功来证明这些付出是有价值的。

当Amazon Web Services（AWS）在2006年出现的时候，它标志着行业的转变。许多以前重复、耗时的任务变得不必要了，启动新服务的成本急剧下降。突然之间，任何有创意和行动能力的人都可以在世界级的基础设施上建立一个全球性的业务，并且只需要付出每小时几美分的成本。就针对已有的格局的颠覆性而言，一些技术明显凌驾于其他领域之上，AWS就是其中之一。

当今，进步的步伐依然不减。2014年11月，在拉斯维加斯举行的年度re:Invent大会上，AWS宣布这一次与会的人数超过13 000人，从2008年起，AWS的主要新功能和服务的数量每年都几乎翻了一倍。因为现有服务也有类似比例的增长，S3和EC2服务较上年有大约100%的增长。这种增长为工程师和企业提供了新的机遇，提供了着力解决一些具有挑战性的问题的能力，可以帮助他们去构建互联网领域一些具有挑战的问题。

不用说，这种前所未有的力量和灵活性以极大的复杂性作为代价。在客户期待并响应客户需求的情况下，AWS已经提供了许多的服务，其中数以千计的功能和特性有时候容易让新用户混淆。这些显而易见的好处也伴随着一个个崭新的词汇和独特架构以及最佳技术实践。当然，有时这些服务功能上的重叠也会使初学者困惑。

本书通过示例让读者掌握知识，揭示了学习AWS面临的挑战。两位作者Andreas和Michael从事于用户可能遇到的重要的服务和功能，并且把安全性的考虑放在了优先和中心的位置。这样有助于建立安全的云中的托管系统，即使是对安全敏感的应用也是安全的。因为许多读者将

会在使用中收到账单，所以书中对需要付费的例子都会明确指出。

作为一名顾问、作者和工程师，我非常赞赏那些对新用户介绍云计算美妙世界的所有努力。我可以自信地说，本书是一本实用指南，可以帮助我们穿过迷宫走向行业领先的云计算平台。

有了这本书作为助手，你会在AWS云上搭建什么？

Ben Whaley

AWS社区英雄

《The UNIX and Linux System Administration Handbook》一书的作者

## 前言

当我们开始开发软件时，我们并不关心运维。我们编写代码，其他人负责部署和运维。这就意味着，软件开发和运维之间存在巨大的鸿沟。更重要的是，发布新功能往往意味着巨大的风险，因为我们无法针对基础设施所有的变更进行手动测试。当需要部署新功能时，每隔6个月我们就要经历一场噩梦。

时间流逝，我们开始负责一个产品。我们的目标是快速迭代，并且能够每周对产品发布新的功能。我们的软件要负责管理资金，因此，软件和基础设施的质量与创新能力一样重要。但是，缺乏灵活性的基础设施和过时的软件部署过程使这一目标根本无法实现。于是，我们开始寻找更好的方法。

我们搜索到了Amazon Web Services（AWS），它为我们提供了灵活、可靠的方式来构建和运行我们设计的应用程序。让我们的基础设施的每一部分实现自动化的可能性令人着迷。逐步地，我们尝试不同的AWS服务，从虚拟服务器到分布式消息队列。能够将运行SQL数据库或在负载均衡器上终止HTTPS连接这类任务外包的特性为我们节省了大量的时间。我们将节省下来的时间投入到实现整个基础设施的自动化测试和运维上。

技术方面并不是在向云转型过程中发生的唯一变化。一段时间后，软件架构从单体应用转变为微服务架构，软件开发与运维之间的鸿沟消失了。相反，我们围绕DevOps的核心原则——“谁构建，谁运维”——构建了我们的组织机构。

我们公司是首家在AWS上进行运营的德国银行。我们在这个云计算的旅程中学到了很多关于Amazon Web Services、微服务以及DevOps的知识。

今天，作为顾问，我们正致力于帮助我们的客户充分利用AWS。有趣的是，他们大多数都不关心怎样节省云计算的成本，相反，他们正在改变他们的组织，从AWS提供的创新空间中获益并领先于自己的竞



争对手。

2015年1月，当我们受邀编写一本关于AWS的书时，感到非常惊讶。但是，在我们与Manning出版社的编辑第一次通电话之后，体会到了Manning出版社的专业水准，我们变得越来越有信心。我们喜欢读书以及传授和分享我们的知识，所以写一本书应该是一个完美的契合。

由于Manning出版社和MEAP读者的巨大支持，我们得以在9个月内完成了本书。我们喜欢我们自己、编辑和MEAP读者之间的循环反馈。可以说创建和改进作为本书一部分的所有示例是非常有趣的。


## 资源与支持

本书由异步社区出品，社区（<https://www.epubit.com/>）为您提供相关资源和后续服务。

## 配套资源

本书提供如下资源：

- 本书源代码。

要获得以上配套资源，请在异步社区本书页面中点击 ，跳转到下载界面，按提示进行操作即可。注意：为保证购书读者的权益，该操作会给出相关提示，要求输入提取码进行验证。

## 提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，点击“提交勘误”，输入勘误信息，点击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的100积分。积分可用于在异步社区兑换优惠券、样书或奖品。

详细信息

写书评

提交勘误

页码:  页内位置 (行数):  勘误印次:

B I U           

字数统计

提交

## 与我们联系

我们的联系邮箱是[contact@epubit.com.cn](mailto:contact@epubit.com.cn)。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问[www.epubit.com/selfpublish/submission](http://www.epubit.com/selfpublish/submission)即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。



## 关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下IT专业图书社区，致力于出版精品IT技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于2015年8月，提供大量精品IT技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网<https://www.epubit.com>。

“异步图书”是由异步社区编辑团队策划出版的精品IT专业图书的品牌，依托于人民邮电出版社近30年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术 etc。



异步社区



微信服务号

## 致谢

写一本书是非常耗时的。我们投入了大量的时间，其他人也一样投入了大量的时间。我们认为时间是地球上最有价值的资源之一，我们要尊重帮助我们完成这本书的人们为此所花费的每一分钟。

感谢所有购买MEAP版本的读者，你们的信任激励我们完成这本书，而且你们还分享了自己对AWS的兴趣。感谢你们阅读这本书，希望你们能有所收获。

感谢所有在本书作者在线论坛上发表评论以及为改进这本书提供精彩反馈的人们。

感谢所有从第一页到最后一页提供了详细评论的审阅者，他们是Arun Allamsetty、Carm Vecchio、Chris Bridwell、Dieter Vekeman、Ezra Simeloff、Henning Kristensen、Jani Karhunen、Javier Muñoz Mellid、Jim Amrhein、Nestor Narvaez、Rambabu Posa、Scott Davidson、Scott M. King、Steffen Burzlaff、Tidjani Belmansour和William E. Wheeler。你们的建议帮助我们塑造了这本书，希望你们像我一样喜欢这本书。

我们也要感谢Manning出版社对我们的信任。这是我们写的第一本书，所以我们知道他们在承担极高的风险。我们要感谢Manning出版社以下工作人员的出色工作。

- Dan Maharry，是你帮助我们在教授AWS的时候不要缺少重要的步骤。感谢你如此耐心，容忍我们同样的错误犯好几次。我们也想感谢Jennifer Stout和Susanna Kline在Dan度假的时候给予的帮助。
- Jonathan Thoms，是你帮助我们思考如何表达代码背后的思想。
- Doug Warren，是你检查了我们的代码示例是否按预期工作。
- Tiffany Taylor，是你完善了我们的英语表达。我们知道你和我们在一起工作会很不容易，因为我们的母语是德语，我们要感谢你的努力。
- Candace Gillhoolley和Ana Romac，是你们帮我们推广这本书。
- Benjamin Berg，是你回答了我们关于写书的许多技术方面的问题。

- Mary Piergies、Kevin Sullivan、Melody Dolab以及所有其他幕后工作者，是你们把粗糙的初稿变成了一本真正的书。

非常感谢Ben Whaley为本书作序。

还要感谢Christoph Metzger、Harry Fix和Tullius Walden Bank团队为我们提供了一个令人难以置信的工作场所，在这里通过将德国第一家银行的原有IT迁移到AWS上，我们获得了很多AWS技能。

最后但同样重要的是，我们要感谢我们生活中的一些重要的人，他们在我们写这本书时默默地支持我们。Andreas要感谢他的妻子Simone，Michael要感谢他的合伙人Kathrin在过去9个月的耐心和鼓励。

## 关于本书

本书介绍了一些重要的AWS服务，以及如何组合它们以便充分利用Amazon Web Services。我们的大多数示例都使用典型的Web应用程序来展示要点。我们非常重视安全话题，所以本书遵循“最小特权”的原则，而且尽可能使用官方的AWS工具。

自动化贯穿于整本书，所以最终读者会很乐意使用自动化工具CloudFormation，来以自动化的方式设置自己所学到的知识，这将是读者能从本书中学到的最重要的技能之一。

本书将介绍3种类型的代码清单：Bash、JSON和Node.js/JavaScript。我们使用Bash创建小型的脚本，以自动方式与AWS进行交互；JSON用于以CloudFormation可以理解的方式描述基础设施；当需要编程来使用服务时，我们使用Node.js平台用JavaScript创建小型应用程序。

我们将Linux作为本书中虚拟服务器的操作系统。所有的示例尽可能地基于开源软件。

## 路线图

第1章介绍云计算和AWS。我们将了解关键概念和基础知识，并创建和设置自己的AWS账户。

第2章将Amazon Web Services带入具体操作中。我们将轻而易举地进入复杂的云基础设施。

第3章是关于使用虚拟服务器的。这一章将借助一些实际的例子，讲解EC2服务的主要概念。

第4章会展示实现自动化基础设施的不同方法。通过使用3种不同的方法，我们将了解何谓“基础架构即代码”，这3种方法分别是终端、编程语言和称为CloudFormation的工具。

第5章会介绍将软件部署到AWS的3种不同方法。我们将使用每种工具以自动方式将应用程序部署到AWS。

第6章是关于安全性的。我们将学习如何使用私有网络和防火墙来保护自己的系统，还将学习如何保护自己的AWS账户。

第7章会介绍了提供对象存储服务的S3，以及提供长期存储服务的Glacier。我们将学习如何将对象存储集成到应用程序中以实现无状态服务器，并用以创建映像库的示例。

第8章是关于AWS提供的虚拟服务器的块存储的。如果计划在块存储上运行原有的软件，这是一个有趣的话题。我们还可以进行一些性能的测量，以了解AWS上可用的选项。

第9章会介绍RDS，这是一种为客户管理关系数据库系统（如PostgreSQL、MySQL、Oracle和Microsoft SQL Server）而提供的服务。如果客户的应用程序使用这种关系数据库系统，这就是实现无状态服务器架构的简单方法。

第10章会介绍DynamoDB，一个提供NoSQL数据库的服务。我们可以将该NoSQL数据库集成到应用程序中以实现无状态服务器。我们将在

这一章中实现一个待办事宜应用的程序。

第11章是关于独立的服务器和完整的数据中心的基础知识的。我们将学习如何在同一个或另一个数据中心的恢复单个EC2实例。

第12章会介绍将系统解耦以增加可靠性的概念。我们将学习如何在AWS上的负载均衡器的帮助下实现同步解耦。异步解耦也是这一章内容的一部分，我们解释如何使用SQS（一种分布式队列服务）搭建容错系统。

第13章会展示如何使用所学的许多服务搭建容错的应用程序。在这一章中，我们将学习基于EC2实例设计容错Web应用程序所需的所有内容，默认情况下它们是不会容错的。

第14章的内容都是关于系统灵活性的。我们将学习如何根据调度或基于系统当前的负载来扩展基础结构的容量。

## 代码的约定和下载

代码清单中或正文中的所有源代码都是等宽字体，以便与普通文本区分开。许多代码清单附带了代码注释，突出了重要的概念。在某些情况下，有编号的项目符号与代码清单后面的说明联系起来，有时我们需要将一行分成两行或更多，以适应页面。在我们的Bash代码中，我们使用了延续的反斜杠。在我们的JSON和Node.js/JavaScript代码中，➡这个符号表示一个人为换行符。

本书中的示例代码可从出版社的官方网站下载。

## 关于作者

安德烈亚斯·威蒂格（Andreas Wittig）和迈克尔·威蒂格（Michael Wittig）作为软件工程师和顾问，专注于AWS以及Web应用程序和移动应用程序开发。他们与遍及全球的客户一同工作。他们一起将德国银行的整个IT基础设施迁移到了AWS。这在德国银行界算是首例。Andreas和Michael在分布式系统开发和架构、算法交易和实时分析方面具有专长。他们是DevOps模型的拥趸，且都是AWS认证的专业级AWS解决方案架构师（AWS Certified Solutions Architect, Professional Level）。



## 关于封面插画

本书封面上的图片标题为“Paysan du Can-ton de Lucerne”，这是瑞士中部卢塞恩州的一名农民。这张照片摘自《Jacques Grasset de Saint-Sauveur》（1757—1810）1797年在法国出售的来自各国的服装服饰的图集《Costume deDifférent Pays》。每幅画都是经过精心绘制和手工上色的。

Grasset de Saint-Sauveur的图集丰富多彩，让我们生动地看到了世界各地的城市和地区在200多年前的文化差异。彼此隔绝，人们讲着不同的方言和语言。在街头或农村，通过人们的服装服饰很容易就能确定他们住在哪里，以及他们的身份和职位。

从那时起，我们的着装方式发生了变化，当时地域的多样化带来的着装上的丰富多彩已经渐渐消逝。现在很难分辨不同地域的居民，更别说不同城镇、不同地区或不同国家的居民了。也许我们已经用更多样化的个人生活交换了文化多样性——当然这是为了一个更多样化和快节奏的科技生活。

在很难分辨出一本计算机相关的图书的时候，Manning出版社以两个世纪前丰富多样的地域生活为基础，借用Grasset de Saint-Sauveur的图画作为书籍的封面，以此赞美计算机行业的创造性和主动性。

## 第一部分 AWS云计算起步

你有没有在Netflix上看过影片，在Amazon.com上买过小玩意，或者今天同步过Dropbox上的文件吗？如果有的话，你就已经在后台使用了Amazon Web Services（AWS）。截至2014年12月，AWS运营了140万台服务器，是云计算市场的大玩家。AWS的数据中心广泛分布于美国、欧洲、亚洲和南美洲。但云计算不只是由硬件和计算能力构成的，软件是每个云计算平台的一部分，能使客户体验到云平台的差异。信息技术的研究机构Gartner将AWS列为云计算基础设施即服务（IaaS）的魔力象限的领导者。算上2015年这已经是第四次了。这是因为，AWS平台上的创新速度以及服务的质量都是非常高的。

本书的第一部分将作为了解AWS的第一步，引导读者了解如何使用AWS来改善IT基础设施。第1章将介绍云计算和AWS，读者将了解关键概念和基础知识。第2章将介绍Amazon Web Service的具体操作，让读者轻松进入复杂的云基础设施。

# 第1章 什么是Amazon Web Services

## 本章主要内容

- Amazon Web Services概述
- 使用Amazon Web Services的益处
- 客户可以使用Amazon Web Services做什么的示例
- 创建以及设置Amazon Web Services账户

Amazon Web Service（AWS）是一个提供Web服务解决方案的平台，它提供了不同抽象层上的计算、存储和网络的解决方案。客户可以使用这些服务来托管网站，运行企业应用程序和进行大数据挖掘。这里提到的术语Web服务，它的含义是可以通过Web界面来控制服务。Web界面可以由机器或人类通过图形用户界面来操作。其中最突出的服务是提供虚拟服务器的EC2，以及提供存储服务的S3。AWS上的服务可以配合工作，客户可以使用它们来复制现有的在企业内部部署的系统，或者从头开始设计新的设置。这些服务按使用付费定价模式收取服务费用。

AWS的客户可以选择不同的数据中心。AWS数据中心分布在美国、欧洲、亚洲和南美洲等。例如，客户可以在日本启动一个虚拟服务器，与在爱尔兰启动虚拟服务器是一样的。这使你能够为世界各地的客户提供全球性的基础设施服务。

所有客户都可以使用的AWS数据中心分布在德国、美国（西部1处、东部2处）、爱尔兰、日本、新加坡、澳大利亚和巴西。<sup>[1]</sup>

<sup>[1]</sup> 截至2018年1月，AWS云在全球18个地区内运营着49个可用区，具体信息可以参考AWS官网介绍。——译者注

### AWS使用了什么样的硬件

AWS没有公开其数据中心所使用的硬件。AWS运行的计算、网络和存储的硬件的规模是巨大的。与使用品牌硬件设备的额外费用相比，它很可能使用商品化的硬件组件以节省成本。硬件故障的处理依靠真实的流程和软件。<sup>[2]</sup>

AWS还使用针对其使用场景而特别开发的硬件。一个很好的例子是英特尔Xeon E5-2666 v3 CPU。这款CPU经过优化为EC2 C4系列的虚拟服务器提供支持。

<sup>[2]</sup> Bernard Golden, “Amazon Web Services (AWS) Hardware,” For Dummies.

从更广泛的意义上讲，AWS就是所谓的云计算平台。

## 1.1 什么是云计算

几乎目前每个IT解决方案都标有云计算或者云。一个时髦的词汇可能有助于产品销售，但在本书中却不适用。

云计算或云是针对IT资源的供应和消费的一个比喻。云中的IT资源对用户来说不直接可见，在这之间有多个抽象的层。云计算提供的抽象级别可能会因虚拟硬件与复杂的分布式系统而有所不同。资源可根据需要大量提供，并按使用付费。

下面是美国国家标准和技术研究所（NIST）对云计算的一个较为正式的定义：

云计算是一种普适的、方便的、按需提供网络访问的可配置的计算资源（如网络、服务器、存储、应用程序和服务）的共享池模型，它能够以最少的管理工作量或与提供者交互的方式快速进行分配和发布。

云计算通常被划分成以下几种类型。

- 公有云 —— 由某一机构、公司管理并对公众开放使用的云计算。
- 私有云 —— 在单个的机构中通过虚拟化共享出来的IT基础设施。
- 混合云 —— 公有云和私有云的混合。

AWS提供的是公有云服务。云计算服务也有多种分类。

- 基础设施即服务（IaaS） —— 提供计算、存储和网络功能等基本资源，使用Amazon EC2、Google Compute Engine和Microsoft Azure虚拟机这一类虚拟服务器。
- 平台即服务（PaaS） —— 提供将定制的应用部署到云上的平台，如AWS Elastic Beanstalk、Google App Engine和Heroku。
- 软件即服务（SaaS） —— 结合了基础设施和软件并且运行在云端，包括Amazon WorkSpaces、Google Apps for Work和Microsoft

Office 365这一类办公应用。

AWS产品阵容包含了IaaS、PaaS和SaaS。让我们更具体地了解一下AWS究竟可以做什么。

## 1.2 AWS可以做什么

客户可以使用一个或多个服务组合在AWS上运行任何应用程序。本节中的示例将让读者了解AWS可以做什么。

### 1.2.1 托管一家网店

John是一家中型电子商务企业的CIO。他的目标是为客户提供一个快速可靠的在线商店。他决定由企业自行管理该网站。3年前他在数据中心租用了服务器。Web服务器处理来自客户的请求，数据库存储商品信息和订单。John正在评估他的公司如何利用AWS的优势将同样的设置运行在AWS上，如图1-1所示。

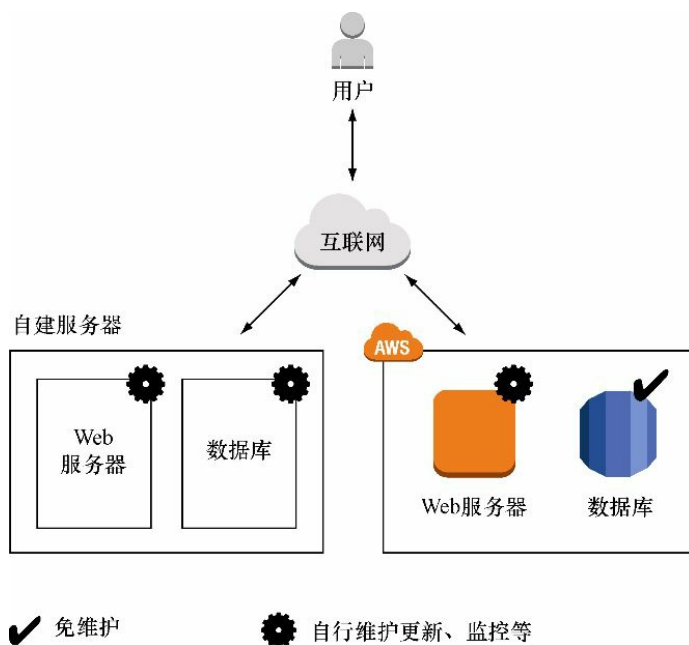


图1-1 运行网店的对比：自建和运行在AWS上

John意识到有其他选择可以通过额外的服务来改进他在AWS上的设置。

- 网店由动态内容（如产品及其价格）和静态内容（如公司标志）等

组成。通过区分动态内容和静态内容，John可以在内容分发网络（CDN）上传递静态内容来减少他的Web服务器的负载并提高性能。

- John在AWS上使用了免维护的服务，包括数据库、对象存储和DNS系统等。这使他免于管理系统的这些部分，降低了运营成本并提高了服务质量。
- 运行网店的应用程序可以安装在虚拟服务器上。John在旧的本地服务器的上配置了多个较小的虚拟服务器，这不需要额外的费用。如果这些虚拟服务器有一台发生故障，负载均衡器将向其他虚拟服务器发送客户请求。这样的配置提高了网站的可靠性。

图1-2展示了John是如何利用AWS增强他们的网店的。

John启动了一个概念验证项目（POC），他发现这些Web应用程序可以迁移到AWS上，并且可以利用AWS上的服务来改进设计。

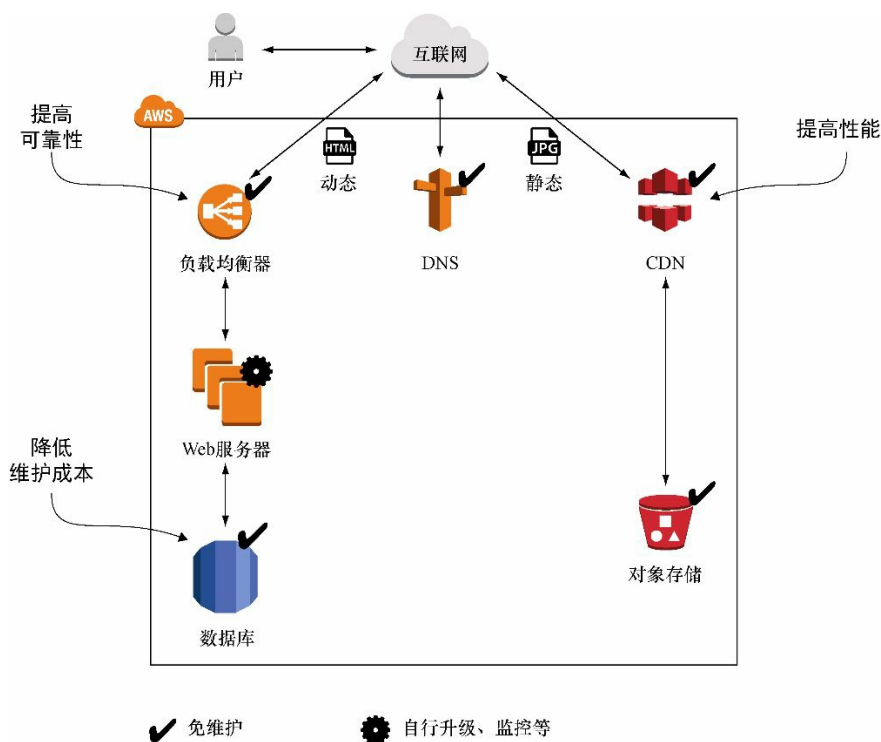


图1-2 在AWS使用CDN使得Web商店获得更好的性能，用负载均衡器实现高可用性，用托管的数据库来降低维护成本

## 1.2.2 在专有网络内运行一个Java EE应用



Maureen是一家全球性企业的高级系统架构师。当公司的数据中心合同在几个月后即将到期的时候，她希望将部分业务应用程序迁移到AWS上，以降低成本并获得灵活性。她发现在AWS上运行企业应用程序是完全可行的。

为此，她在云中设定了一个虚拟网络，并通过虚拟专网（VPN）将其连接到网络。公司可以通过使用子网以及控制列表来管理网络流量，这样就可以满足访问控制和保护关键任务的数据。Maureen使用网络地址转换（NAT）和防火墙来控制互联网的流量。她将应用程序服务器安装在虚拟机（VM）上以运行Java EE应用程序。Maureen还考虑将数据存储在SQL数据库服务之中（如Oracle数据库企业版或Microsoft SQL Server EE版）中。图1-3解释了Maureen的架构。

Maureen已经成功地将本地数据中心与AWS上的私有网络连接起来。她的团队已经开始将第一个企业应用程序迁移到云端。

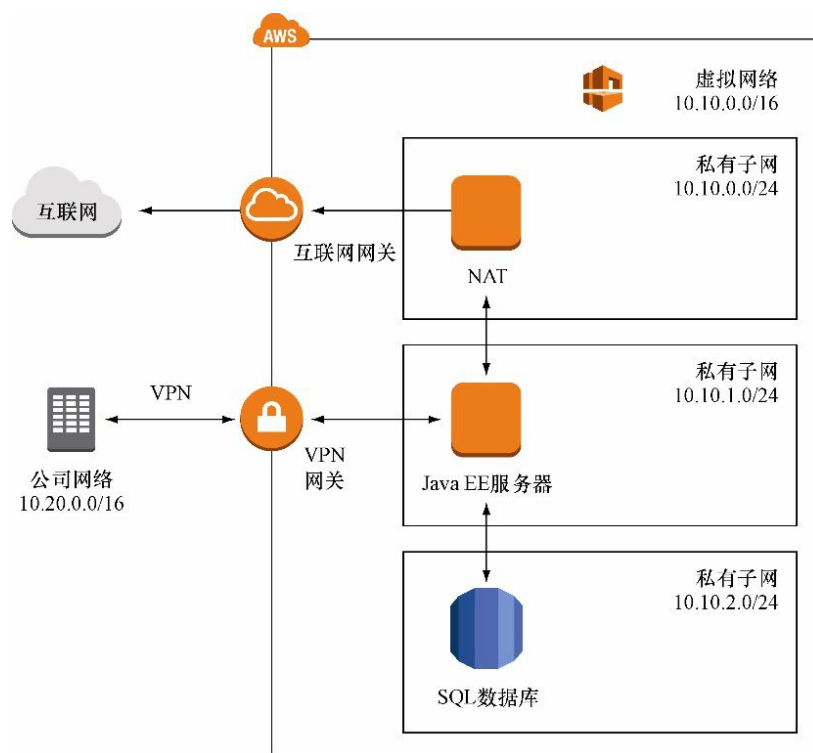


图1-3 在企业网络和AWS的环境中运行Java EE应用

### 1.2.3 满足法律和业务数据归档的需求

Greg负责管理一个小型律师事务所的IT基础设施。他的主要工作目标是以可靠、耐用的方式存储和归档所有数据。他运行了一个文件服务器，提供了在办公室内共享文件的功能。存储所有数据对他来说是一个挑战。

- 他需要备份所有文件，以防止关键数据丢失。为此，Greg将数据从文件服务器复制到另一个网络附加存储（network-attached storage, NAS）上。为此他不得不再一次为文件服务器购买硬件。文件服务器和备份服务器的位置距离很近，因此他无法满足灾难恢复的要求，如从火灾或突发事件中恢复数据。
- 为了满足法律和业务数据存档的需求，Greg需要能够长时间地存储数据。将数据存储10年或更长时间是件很棘手的事情。Greg使用了一个昂贵的归档解决方案。

为了节省资金并提高数据安全性，Greg决定使用AWS。他将数据传输到具有高可用特性的对象存储服务上。存储网关使得不需要购买和管理本地的网络附加存储和本地备份。一个虚拟的磁带机负责完成在所需的时间段内归档数据的任务。图1-4展示了Greg如何在AWS上实现这个使用场景，并将其与本地的解决方案进行比较。

Greg在AWS上存储和归档数据的新的解决方案很棒，因为能够明显地提高质量，并获得了扩展存储大小的可能性。

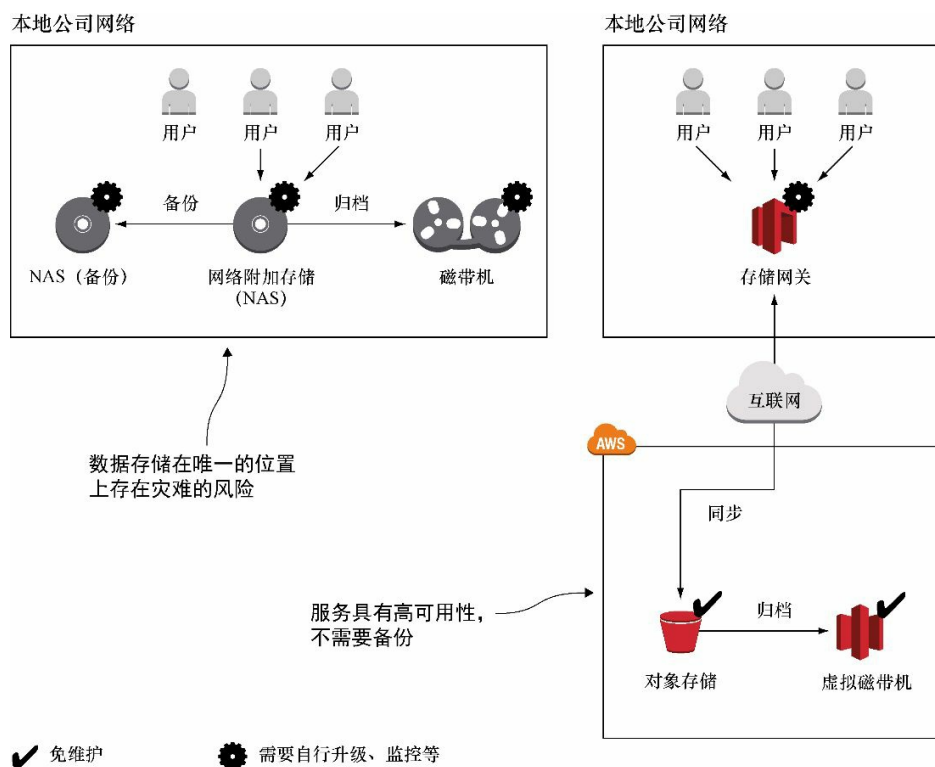


图1-4 在专有环境和AWS下备份和归档数据

## 1.2.4 实现容错的系统架构

Alexa是一名软件工程师，工作于一家创业企业。她知道墨菲定律适用于IT基础设施：任何可能出错的事情都会出错。Alexa正在努力构建一个容错系统，以防止运行中业务出现中断。她知道AWS上有两种类型的服务：容错服务和可以以容错方式运行的服务。Alexa构建了一个如图1-5所示的具有容错架构的系统。数据库服务提供复制和故障转移处理。Alexa使用了虚拟服务器充当Web服务器，这些虚拟服务器默认情况下不具有容错的特性。但是，Alexa使用了负载均衡器，并可以在不同的数据中心启动多台服务器以实现容错。

到目前为止，Alexa采用的方法已经在事故中保护了公司的系统。不过，她和她的团队总是在为各种系统失效做计划。

现在读者对AWS可以做什么应该有了一个广泛的了解。一般来说，客户可以在AWS上托管任何应用程序。下一节将介绍AWS提供的9个重要的好处。

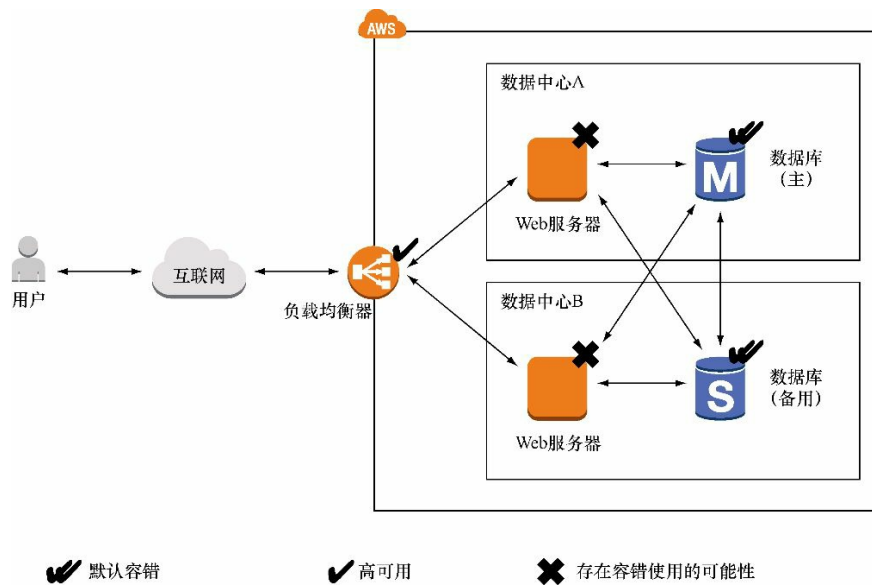


图1-5 在AWS上构建容错系统

## 1.3 如何从使用AWS上获益

使用AWS最重要的优势是什么？你可能会说是节省成本。但省钱肯定不是唯一的优势。让我们看看你可以从使用AWS中获益的其他方法。

### 1.3.1 创新和快速发展的平台

2014年，AWS在拉斯维加斯举办的年度会议re:Invent上宣布了500多项新的服务和功能。除此之外，几乎AWS每周都会有新功能和改进的发布。客户可以将这些新的服务和功能转化为针对自己的客户的创新解决方案，从而体现竞争优势。

re:Invent大会的与会者的数量从2013年的9 000人增加到2014年的13 500人。<sup>[3]</sup> AWS的客户中有超过100万家企业和政府机构，在2014年第一季度的业绩沟通会上，该公司表示将继续聘请更多的人才谋求进一步的发展。<sup>[4]</sup> 在未来几年里，我们可以期待更多的新功能和新服务。

<sup>[3]</sup> Greg Bensinger, “Amazon Conference Showcases Another Side of the Retailer’s Business,” *Digits*, Nov. 12, 2014.

<sup>[4]</sup> “Amazon.com’s Management Discusses Q1 2014 Results - Earnings Call Transcript,” *Seeking Alpha*, April 24, 2014.

### 1.3.2 解决常见问题的服务

如我们所了解的，AWS是一个服务平台。常见的问题，如负载均衡、队列、发送电子邮件以及存储文件等，都可以通过服务加以解决。而客户不需要“重新发明轮子”。客户的工作就是选择合适的服务来构建复杂的系统。然后，客户可以让AWS来管理这些服务，而自己则可以专注于所服务的客户。

### 1.3.3 启用自动化

由于AWS提供了API，因此客户可以自动执行所有的操作：客户可以编写代码来创建网络，启动虚拟服务器集群或部署关系数据库。自动化提高了可靠性，并提高了效率。

系统拥有的依赖性越大，它就会变得更复杂。面对复杂的图形，人类可能很快就失去透视能力，而计算机可以应付任何大小的图形。客户应该集中精力于人类擅长的领域——描述系统的任务，而计算机则会了解如何解决所有这些依赖关系来创建系统。基于客户的蓝图在云平台上设置所需的环境可以通过基础设施即代码以自动化的方式来完成，关于这部分内容将在第4章介绍。

### 1.3.4 灵活的容量（可扩展性）

灵活的容量的特性可以使客户免于做规划。客户可以从一台服务器扩展到数千台服务器。客户的存储容量可以从GB级别增长到PB级别。客户不再需要预测未来几个月和几年的容量需求。

如果你经营一家网店，则会有季节性流量的模式，如图1-6所示。想想白天与晚上、平日与周末或假期。如果你可以在流量增长时增加容量并在流量缩减时减少容量，那岂不是很好吗？这正是灵活容量的特性。你可以在几分钟之内启动新的服务器，然后在几小时后删除它们。

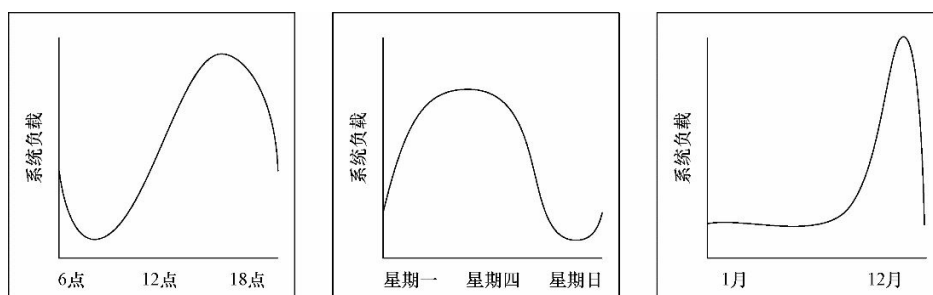


图1-6 网店的季节性流量模式

云计算几乎没有容量的限制。客户不再需要考虑机架空间、交换机和电源供应，客户可以添加尽可能多的服务器。如果数据量增长，则始

终可以添加新的存储容量。

灵活的容量也意味着客户可以关闭未使用的系统。在我们最近的一个项目中，测试环境只在工作日从上午7:00到下午8:00运行，这让我们成本节省了60%。

### 1.3.5 为失效而构建（可靠性）

大多数AWS服务都具有容错或高可用的特性。如果客户使用这些服务，可以免费获得可靠性。AWS支持客户以可靠的方式构建系统，它为客户提供了创建自己的容错系统所需的一切资源。

### 1.3.6 缩短上市的时间

在AWS中，客户请求一个新的虚拟服务器。几分钟后，该虚拟服务器将被启动并可以使用。同样的情况也适用于任何其他AWS服务。客户可以按需使用它们。这使客户能够快速地将自己的基础架构调整以满足新的需求。

由于反馈循环更短，客户的开发过程将更快。客户可以消除各种限制，如可用的测试环境数量。如果客户需要一个更多资源的测试环境，可以创建并运行数小时。

### 1.3.7 从规模经济中受益

在编写此书时，自2008年以来使用AWS的费用已经降至最初的1/42。

- 在2014年12月，出站数据的传输费用降低了43%。
- 在2014年11月，使用搜索服务的费用降低了50%。
- 在2014年3月，使用虚拟服务器的费用降低了40%。

截至2014年12月，AWS运营了140万台服务器。所有与操作有关的

过程都必须进行优化，以便在如此规模上运行。AWS规模越大，价格就会越低。<sup>[5]</sup>

<sup>[5]</sup> 截至2017年11月，AWS已经进行了62次降价。——译者注

## 1.3.8 全球化

客户可以将应用程序部署到尽可能接近自己的客户的地方。AWS在以下位置建有数据中心：

- 美国（弗吉尼亚北部、加利福尼亚北部、俄勒冈）；
- 欧洲（德国、爱尔兰）；
- 亚洲（日本、新加坡）；
- 澳大利亚；
- 南美（巴西）。

有了AWS，客户就可以在全球运营自己的业务。<sup>[6]</sup>

<sup>[6]</sup> 除上述之外，AWS在全球部署基础设施的国家和地区还包括了美国（俄亥俄）、印度（孟买）、韩国（首尔）、加拿大、欧洲（法兰克福、伦敦、巴黎）、中国（北京、宁夏）等。——译者注

## 1.3.9 专业的合作伙伴

AWS符合以下合规性要求。

- ISO 27001 ——全球性信息安全标准，由独立机构认证。
- FedRAMP & DoD CSM ——美国联邦政府和美国国防部云计算安全标准。
- PCI DSS Level 1 ——支付卡行业（PCI）的数据安全标准（DSS），用以保护持卡人的数据安全。
- ISO 9001 ——全球范围内使用的标准化质量管理方法，并由独立和认证机构认证。

如果你还不相信AWS是专业的合作伙伴，那么你应当知道AWS已



经承担了Airbnb、Amazon、Intuit、NASA、Nasdaq、Netflix和SoundCloud等重要的任务。

在下一节我们将详细阐述成本效益。

## 1.4 费用是多少

AWS的账单类似于电费账单。服务根据用量收费。客户需要支付运行虚拟服务器的时间、从对象存储库使用的存储空间（以GB为单位）或正在运行的负载均衡器的数量。服务按月开具发票。每项服务的定价是公开的，如果要计算计划中每月的成本，可以使用“AWS简单月度计算器”（AWS Simple Monthly Calculator）来估算。

### 1.4.1 免费套餐

在注册后的前12个月内客户可以免费使用一些AWS服务。免费套餐的目的是让客户能够对AWS进行实验并获得一些经验。下面是免费套餐中包含的内容。

- 每月运行Linux或者Windows小型虚拟服务器750 h（大约1个月）。这意味着客户可以整个月运行一个虚拟服务器，也可以同时运行750个虚拟服务器1 h。
- 每月750 h（大约1个月）的负载均衡器。
- 具有5 GB存储空间的对象存储。
- 具有20 GB存储空间的小型数据库，包括备份。

如果超出了免费套餐的限制，就要开始为使用的资源支付费用，而不再另行通知。客户将在月底收到一张账单。在开始使用AWS之前，我们将向读者展示如何监控成本。如果免费套餐在一年后结束，读者将为所使用的所有资源付费。

客户还可以获得一些额外的好处，详情参见<http://aws.amazon.com/free>。本书尽可能多使用免费套餐的资源，并清楚地说明何时需要额外的不在免费套餐之内的资源。

### 1.4.2 账单样例

如前所述，客户可能会被以下几种方式计费。

- 按使用时间计费 ——如果客户使用服务器的时间是61 min，通常会计算为2 h<sup>[7]</sup>。

<sup>[7]</sup> AWS 从2017年10月2日起以按需、预留和竞价形式发布的Linux实例的使用将按1 s的增量计费。同样，EBS卷的预置存储也将按1 s的增量计费。——译者注

- 按流量计费 ——流量以吉字节（GB）或者请求数量来衡量。
- 按存储用量计费 ——可以按照配置的容量（例如，50 GB的卷，不管使用了多少）或者实际用量（例如，使用了2.3 GB）。

还记得在1.2节提过的网店的例子吗？图1-7展示了这家网店使用AWS的示意图，并添加了有关各个部分的计费信息。

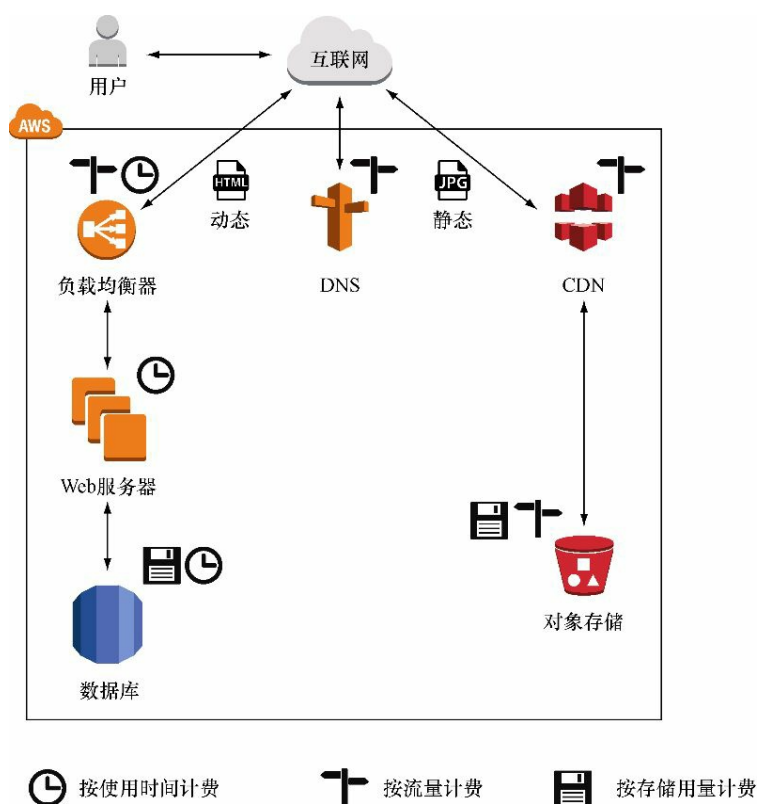


图1-7 网店的例子

假设你的网店在1月开始启动，并且你决定开展营销活动以提高下个月的销售额。幸运的是，你可以在2月将网店的访问人数增加5倍。正

如你所了解的，你必须根据使用情况支付AWS。表1-1展示了你的1月和2月的账单。访客人数从10万人增加到50万人，月收入从142.37美元增加到538.09美元，涨幅是3.7倍。因为你的网店必须处理更多流量，所以你必须为更多的服务（如CDN、Web服务器和数据库）付费。其他服务，如静态文件的存储，因为没有增加更多的用量，所以价格保持不变。

表1-1 如果网店访客的数量增加，AWS的账单将如何变化

服 务	1月用量	2月用量	2月费用 (美元)	增加（美 元）
网站访客	10万人	50万人		
CDN	2 600万条请求， 25 GB流量	13 100 万条请求 125 GB流量	113.31	90.64
静态文件	使用50 GB的存储	使用50 GB的存储	1.50	0.00
负载均衡器	748 h+50 GB流量	748 h+250 GB流量	20.30	1.60
Web服务器	1台服务器=748 h	4台服务器=2 992 h	204.96	153.72
数据库（748 h）	小型服务器+20 GB存储	大型服务器+20 GB 存储	170.66	128.10
流量（出站到互联网流量）	51 GB	255 GB	22.86	18.46
DNS	200万条请求	1 000万条请求	4.50	3.20
总成本			538.09	395.72

使用AWS，客户可以实现流量和成本之间的线性关系，而其他机会正等待客户使用这个定价模式。

### 1.4.3 按使用付费的机遇

AWS按使用付费的定价模式创造了新的机会。客户不再需要对基础设施进行前期投资。客户可以根据需要启动服务器，并且只支付每小时使用时间的费用，客户可以随时停止使用这些服务器，而不必再为此付费。客户不需要对自己将使用多少存储进行预先承诺。

一台大型服务器的成本大致与两个较小的服务器之和相同。因此，客户可以将系统分成几个较小的部分，因为服务器成本是相同的。这种容错的能力不仅适用于大公司，而且还可用于较小预算的场景。

## 1.5 同类对比

AWS不是唯一的云计算提供商。微软和谷歌也有云计算的产品。

OpenStack则有不同，因为它是开源的，由包括IBM、HP和Rackspace在内的200多家公司共同开发。这些公司中的每一家都使用OpenStack来运行自己的云计算产品，有时候这些公司会用到闭源的附件。你可以根据OpenStack来运行自己的云，但是将失去1.3节中所描述的大部分好处。

在云计算供应商之间进行比较并不是一件容易的事，因为缺少太多的开放标准。像虚拟网络和消息队列这样的功能实现的方式差异很大。如果客户知道自己需要什么具体功能，可以比较细节并做出决定；否则，AWS将是客户最好的选择，因为在找到能够解决自己的问题的方法中，AWS的机会是最高的。

以下是云计算服务提供商一些常见的功能：

- 虚拟服务器（Linux和Windows）；
- 对象存储；
- 负载均衡器；
- 消息队列；
- 图形用户界面；
- 命令行接口。

更有趣的一个问题是，云计算服务提供商有何不同？表1-2比较了AWS、Azure、Google Cloud Platform和OpenStack。

表1-2 AWS、Microsoft Azure、Google Cloud Platform和OpenStack的不同

	AWS	Azure	Google Cloud Platform	OpenStack
服务的数量	大部分	一些	满足	少数

位置分布的数量（每个位置有多个数据中心）	9	13	3	有（依赖OpenStack服务提供者）
合规性	公共的标准（ISO 27001、HIPAA、FedRAMP、SOC），IT Grundschutz（德国）、G-Cloud（英国）	公共的标准（ISO 27001、HIPAA、FedRAMP、SOC），ISO 27018（云隐私）、G-Cloud（英国）	公共的标准（ISO 27001、HIPAA、FedRAMP、SOC）	有（依赖Open Stack服务提供者）
SDK语言	Android、浏览器（JavaScript）、iOS、Java、.NET、Node.js（JavaScript）、PHP、Python、Ruby、Go	Android、iOS、Java、.NET、Node.js（JavaScript）、PHP、Python、Ruby	Java、浏览器（JavaScript）、.NET、PHP、Python	—
与开发过程的融合	中级，与特定的生态无关	高级，与Microsoft生态链接（如.NET开发）	高级，与Google生态链接（如Android）	—
块存储（连接的网络存储）	有	有（可同时被多个虚拟服务器使用）	没有	有（可同时被多个虚拟服务器使用）
关系数据库	有（MySQL、PostgreSQL、Oracle Data Base、Microsoft SQL Server）	有（Azure SQL Data Base、Microsoft SQL Server）	有（MySQL）	有（依赖OpenStack服务提供者）
NoSQL数据库	有（专有）	有（专有）	有（专有）	没有
DNS	有	没有	有	没有

虚拟网络	有	有	没有	有
发布/订阅消息服务	有（专有，JMS库可用）	有（专有）	有（专有）	没有
机器学习工具	有	有	有	没有
部署工具	有	有	有	没有
私有数据中心集成	有	有	有	没有

在我看来，AWS是目前可用的、成熟的云计算平台。



## 1.6 探索AWS服务

用于计算、存储和联网的硬件是AWS云计算的基础。AWS在硬件上运行软件服务来提供云服务，如图1-8所示。Web界面、API充当AWS服务和应用程序之间的接口。

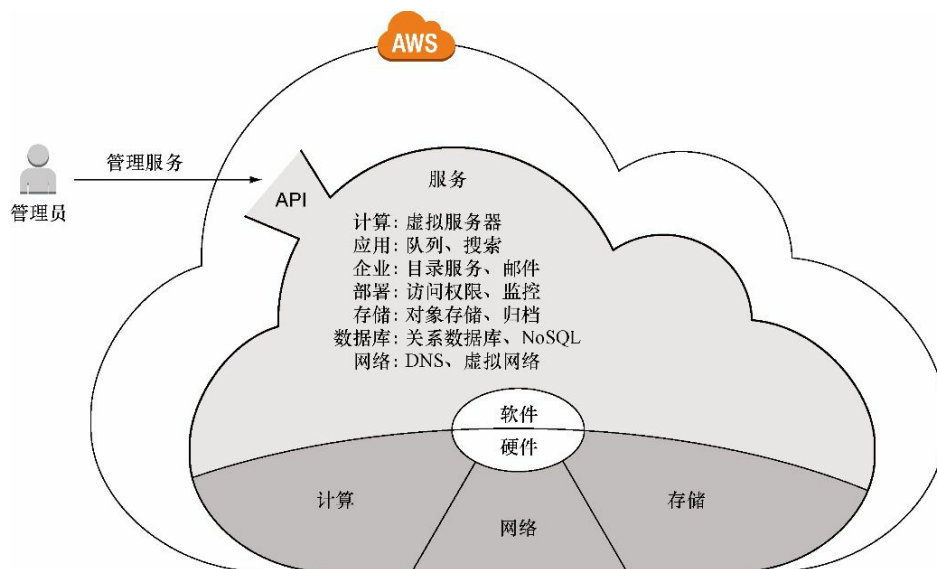


图1-8 AWS云由可通过API访问的硬件和软件服务组成

客户可以通过图形用户界面、使用SDK以编程的方式手动发送请求至API，以实现对服务的管理。为此，客户可以使用诸如管理控制台、基于Web的用户界面或命令行工具等工具。虚拟服务器有其特殊性，例如，客户可以通过SSH连接到虚拟服务器，并获得管理员访问权限。

这意味着客户可以在虚拟服务器上安装所需的任何软件。其他服务，如NoSQL数据库服务则是通过API提供其功能，细节被隐藏到幕后。图1-9展示了管理员在虚拟服务器上安装定制的PHP Web应用程序，并管理所依赖的服务，如PHP Web应用程序使用的NoSQL数据库。

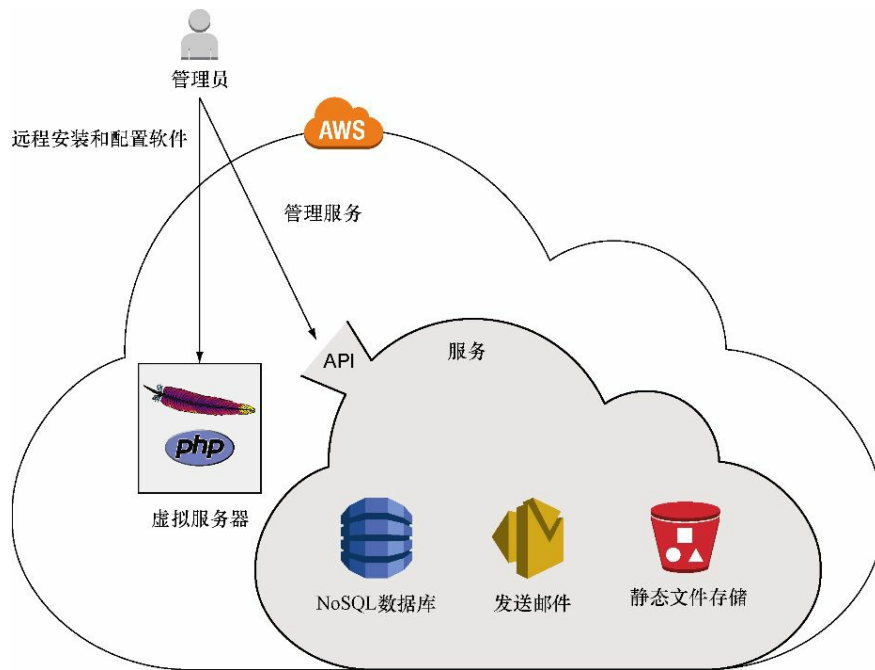


图1-9 管理运行在虚拟服务器以及所依赖的服务上的定制应用

用户将HTTP请求发送到虚拟服务器。在此虚拟服务器有安装Web服务器与定制的PHP Web应用程序。Web应用程序需要与AWS服务进行通信，以便响应用户的HTTP请求。例如，Web应用程序需要从NoSQL数据库查询数据、存储静态文件和发送电子邮件。Web应用程序和AWS服务之间的通信由API处理，如图1-10所示。

一开始，客户可能会惊讶于AWS提供的服务的数量。下面对AWS服务的分类将有助于客户找到所需要的服务。

- 计算服务 提供了计算能力以及内存。客户可以启动虚拟服务器并使用它们来运行应用程序。
- 应用服务 为常见的使用场景提供解决方案，如消息队列、主题以及检索大量数据以集成到应用中。
- 企业服务 提供独立的解决方案，如邮件服务器和目录服务。
- 部署和管理服务 作用于迄今所提到的服务。这个服务可以帮助客户授予或者撤销对云资源的访问、虚拟服务系统的监控以及部署应用程序。
- 存储服务 被用来搜集、保存和归档数据。AWS提供了不同的存储选项：对象存储或者用于虚拟服务器的网络附加存储方案。
- 当需要管理结构化数据时，数据库存储 比其他存储方案有一些优

势。AWS提供了关系数据库和NoSQL数据库服务。

- 网络服务 是AWS的基本组成部分。客户可以定义一个私有的网络并使用高度集成的DNS服务。

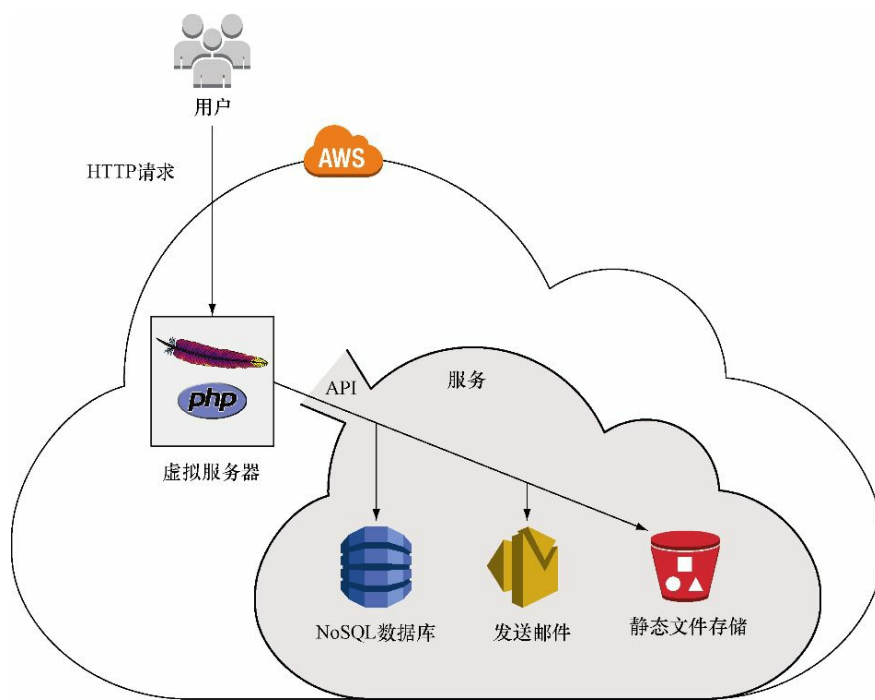


图1-10 定制的Web应用使用AWS服务来处理HTTP请求

要知道我们所列出的仅仅是最重要的服务的类别。其他服务当然也是可用的，同样可以支持客户运行自己的应用程序。

现在我们看一看AWS服务的细节，了解一下如何与这些服务进行交互。

## 1.7 与AWS交互

当客户与AWS进行交互以配置或者使用AWS服务的时候，客户就会调用API。这里提到的API是AWS的入口，如图1-11所示。

接下来，我将给读者提供调用API的可用工具的全貌。读者可以比较这些工具的能力，它们可以帮助你自动完成每日的工作。

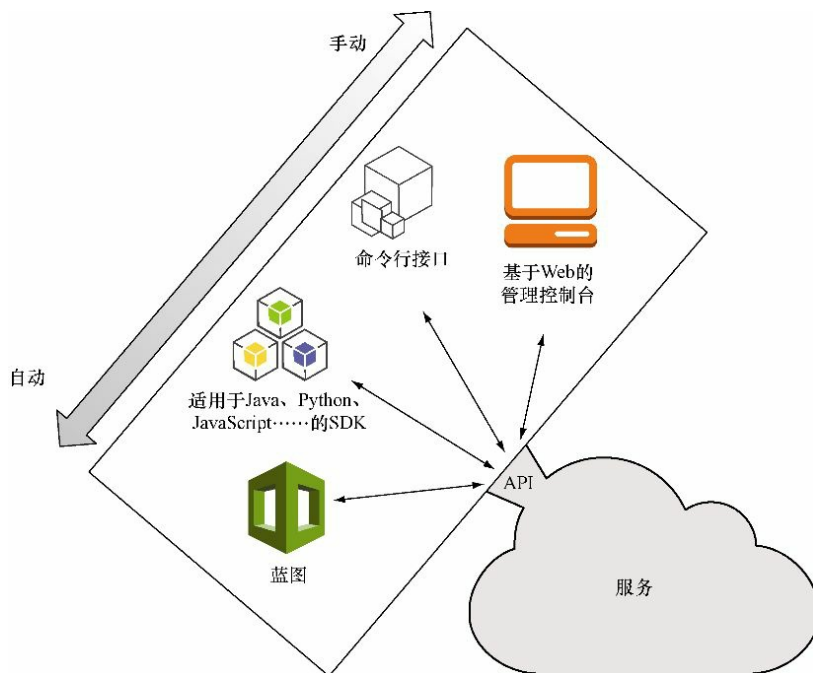


图1-11 与AWS API交互的工具

### 1.7.1 管理控制台

客户可以使用基于Web的管理控制台实现与AWS的交互。通过这个方便的图形用户界面，客户可以手动控制AWS。这个管理控制台支持每一种现代Web浏览器（Chrome、Firefox、Safari 5以上版本、IE 9以上版本），如图1-12所示。

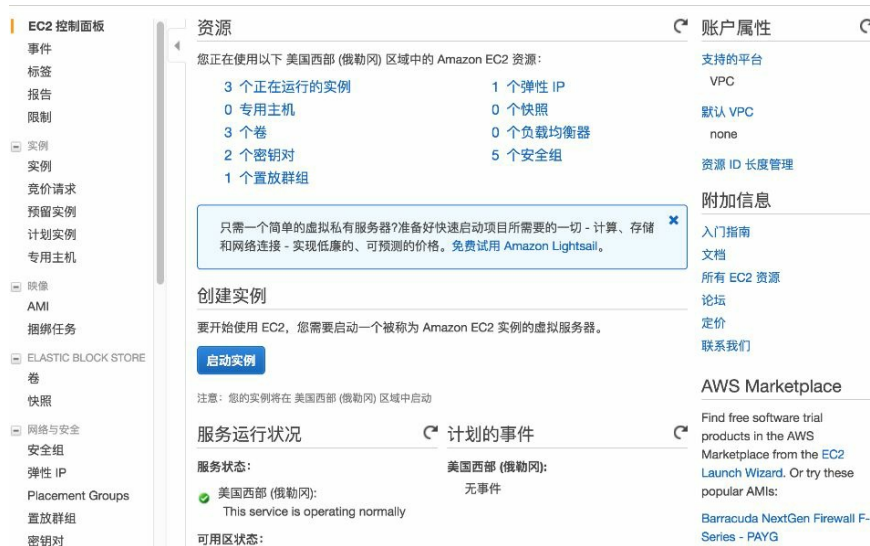


图1-12 管理控制台

如果你正在尝试AWS，那么这个管理控制台就是最好的起点。它能够帮你获得不同服务的全貌并快速取得成功。管理控制台也是为了开发、测试而设置云基础设施的比较好的方式。

## 1.7.2 命令行接口

客户可以通过命令行启动虚拟服务器、设置存储并且发送邮件。使用这个命令行接口（CLI），可以控制AWS的一切，如图1-13所示。

```
michael — bash — 92x39
Last login: Fri Feb 20 09:32:45 on ttys000
mwittig:~ michael$ aws cloudwatch list-metrics --namespace "AWS/EC2" --max-items 3
{
  "Metrics": [
    {
      "Namespace": "AWS/EC2",
      "Dimensions": [
        {
          "Name": "InstanceId",
          "Value": "i-ed62dc0b"
        }
      ],
      "MetricName": "StatusCheckFailed_Instance"
    },
    {
      "Namespace": "AWS/EC2",
      "Dimensions": [
        {
          "Name": "InstanceId",
          "Value": "i-ed62dc0b"
        }
      ],
      "MetricName": "StatusCheckFailed"
    },
    {
      "Namespace": "AWS/EC2",
      "Dimensions": [
        {
          "Name": "InstanceId",
          "Value": "i-0a02beec"
        }
      ],
      "MetricName": "CPUUtilization"
    }
  ],
  "NextToken": "None___3"
}
mwittig:~ michael$
```

图1-13 命令行接口

CLI通常用于自动执行AWS上的任务。如果客户想通过持续集成服务器（如Jenkins）的帮助自动化基础设施的某些部分，则CLI是该任务的正确工具。CLI提供了访问API的便捷方式，并可以将对API的多个调用整合到一个脚本中。

客户甚至可以通过将多个CLI调用链接起来，以实现基础设施的自动化。CLI可用于Windows、Mac和Linux，还有一个适用于PowerShell的版本。

### 1.7.3 SDK

有时客户需要从自己的应用程序中调用AWS。使用SDK，客户可以使用自己喜欢的编程语言将AWS集成到应用程序逻辑中。AWS为以下环境提供了SDK：

- Android;
- Node.js（JavaScript）；

- 浏览器（JavaScript）；
- PHP；
- iOS；
- Python；
- Java；
- Ruby；
- .NET；
- Go<sup>[8]</sup>。

<sup>[8]</sup> 目前也有C++的SDK。——译者注

SDK通常用于将AWS服务集成到应用程序中。如果客户正在进行软件开发，并希望集成AWS服务（如NoSQL数据库或推送通知服务），那么SDK就是该任务的正确选择。某些服务（如队列和主题订阅）必须在应用程序中使用SDK。

## 1.7.4 蓝图

蓝图是包含所有服务和依赖关系的对于系统的描述。蓝图并没有说明实现所描述的系统所必需的步骤或顺序。图1-14展示了如何将蓝图转移到正在运行的系统中。

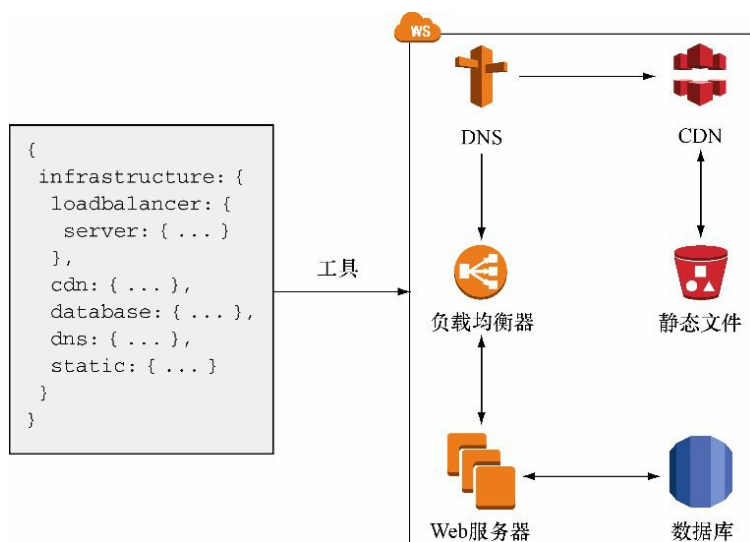


图1-14 使用蓝图实现基础设施自动化

如果你必须控制许多或复杂的环境，可以考虑使用蓝图。蓝图将帮你自动化云中基础设施的配置。例如，你可以使用蓝图来设置虚拟网络并在该网络中启动不同的服务器。

蓝图免除了客户的大部分工作负担，因为客户不再需要担心系统创建期间的依赖关系——蓝图将整个流程自动化。我们将在第4章中了解有关自动化基础设施的更多信息。

现在是开始创建自己的AWS账户并在所有这些理论之后探索AWS实践的时候了。



## 1.8 创建一个AWS账户

在开始使用AWS之前，客户需要创建一个账户。AWS账户是客户拥有的所有资源的一个篮子。如果多个人需要访问该账户，客户可以将多个用户添加到一个账户下面。默认情况下，客户的账户将有一个root用户。要创建一个账户，客户需要提供以下内容：

- 一个电话号码，以验证客户的身份；
- 一张信用卡，以支付客户的账单。

### 使用原有账户可以吗

在使用本书中的示例时，读者可以使用现有的AWS账户。在这种情况下，使用可能不在免费套餐的范围之内，可能需要支付费用。

此外，如果读者在2013年12月4日之前创建了现有的AWS账户，那么应该创建一个全新的AWS账户，否则在尝试运行本书的示例时可能会遇到遗留问题的麻烦。

### 1.8.1 注册

注册的流程包括以下5个步骤。

- (1) 提供登录凭据。
- (2) 提供联系信息。
- (3) 提供支付信息的细节。
- (4) 验证身份。
- (5) 选择支持计划。

将所使用的浏览器指向AWS官方网站，然后单击“创建免费账户”按钮。

## 1. 提供登录凭据

注册页面，为客户提供了两个选择，如图1-15所示。客户可以使用 Amazon.com 账户创建账户，也可以重新开始创建账户。如果创建新账户，请按照下面的步骤；否则，请跳到第5步。

填写电子邮件地址，点击“继续”，创建登录凭据。我们建议选择一个强大的密码来防止误用。

我们建议密码的长度为16字节，包括数字和符号。如果有人可以访问你的账户，这就意味着他们有可能破坏你的系统或者窃取你的数据。



图1-15 创立一个AWS账号：注册页

## 2. 提供联系信息

下一步需要提供联系信息，如图1-16所示。填写完所需的全部内容，然后继续下一步。

账户类型 ⓘ

(1) ☒ 专业级 ☐ 个人

全名  
yunjisuan

公司名称  
yunjisuan company

电话号码  
[REDACTED]

国家/地区  
中国

地址  
No.10 Jiuxianqiao Road  
Chaoyang District

城市  
Beijing

州、省或地区  
Chaoyang

邮政编码  
100085

(2) ☒ 选中此框表示您已阅读并同意 [AWS 客户协议](#) 的条款

创建账户并继续

图1-16 创建一个AWS账户：提供联系信息

### 3. 提供支付信息的细节

现在的屏幕显示如图1-17所示，需要支付信息。AWS支持MasterCard以及Visa信用卡。如果不想以美元支付自己的账单，可以稍后再设置首选付款货币。这里支持的货币有欧元、英镑、瑞士法郎、澳元等。

付款信息

请输入您的付款信息，以便我们验证您的身份。除非您的使用超出了 [AWS 免费层次限制](#)，否则我们不会收取费用。请查看[常见问题](#)了解更多信息。

(1)

信用卡号/借记卡号

到期日期

11 2019

持卡人姓名

(2)

账单地址

☒ 使用我的联系人地址

chaoyang district no.10 jiuqiaoqian road  
beijing haidian 100085  
CN

☐ 使用新地址

安全提交

图1-17 创建一个AWS账户：提供支付信息的细节

## 4. 验证身份

接下来就是验证身份。图1-18展示了这个流程的第一步。

当完成这个部分以后，你会接到一个来自AWS的电话呼叫。一个机器人的声音会询问你的PIN码，这个PIN码将会如图1-19所示显示在屏幕上。身份被验证以后，就可以继续执行最后一步。

电话验证

AWS 将立即使用自动系统给您打电话。出现提示时，从手机键盘上的输入 AWS 网站出现的 4 位数字。

**提供电话号码**  
请在下方输入您的信息，然后单击“立即呼叫我”按钮。

国家/地区代码  
中国 (+86)

电话号码 分机

(1) [Redacted]

安全性检查

687n77

(2) [Redacted]

(3) 立即呼叫我

图1-18 创建一个AWS账户：身份验证（1/2）

正在呼叫...

请回答来自 AWS 的电话，并在出现提示时在手机键盘上输入 4 位数字。

2 0 [Redacted] [Redacted]

立即呼叫我

图1-19 创建一个AWS账户：身份验证（2/2）

## 5. 选择支持计划

最后一步就是选择支持计划，如图1-20所示。在这种情况下，请选择免费的“基本方案”。如果你以后为自己的业务创建一个AWS账户，我

们建议你选择AWS的“业务方案”。你甚至可以以后切换支持计划。



图1-20 创建一个AWS账户：选择支持计划

现在你已经完成了全部步骤，可以使用AWS管理控制台登录到自己的账户了。

## 1.8.2 登录

你现在已经有了一个AWS账户，可以登录AWS管理控制台。如前所述，管理控制台是一个基于Web的工具，可用于控制AWS资源。

管理控制台使用AWS API来实现客户需要的大部分功能。图1-21展示了登录页面。

输入你的登录凭据，然后点击“下一步”，就可以看到图1-22所示的管理控制台。



图1-21 登录到管理控制台

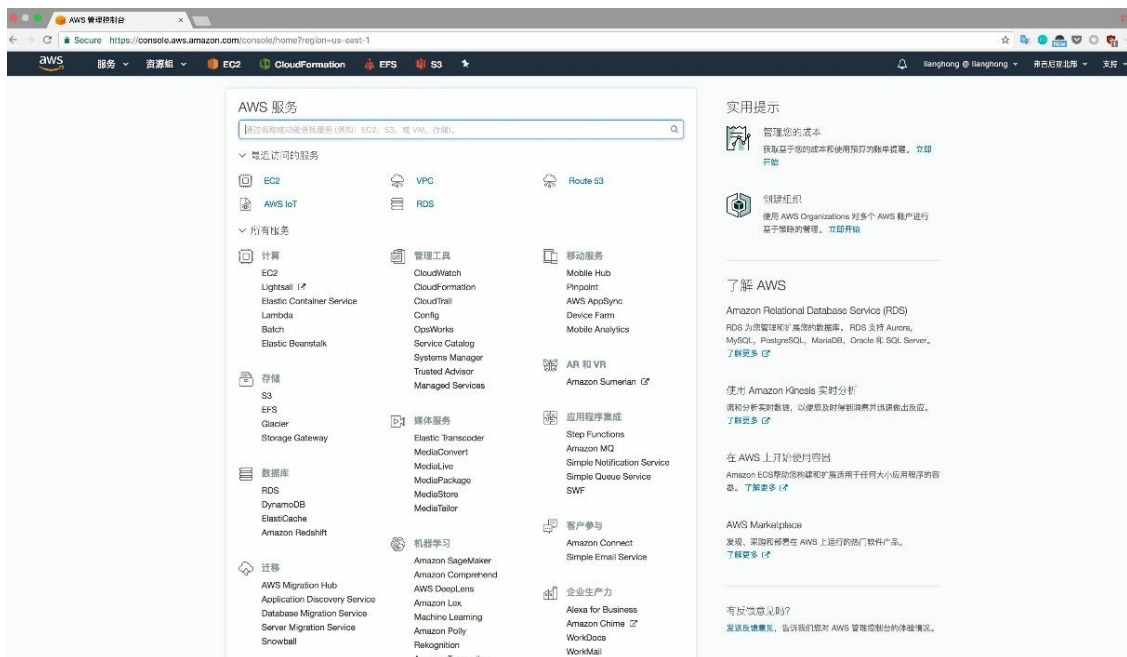




图1-22 AWS管理控制台

在这个页面中最重要的部分是顶部的导航栏，如图1-23所示。它由以下6个部分组成。

- AWS ——提供一个账户中全部资源的快速概览。
- 服务 ——提供访问全部的AWS服务。
- 自定义部分 ——点击“编辑”并拖放重要的AWS服务到这里，实现个性化的导航栏。
- 客户的名字 ——让客户可以访问账单信息以及账户，还可以让客户退出。
- 客户的区域 ——让客户选择自己的区域，我们将在3.5节介绍“区域”的概念，现在不需要改变任何内容。
- 支持 ——让客户可以访问论坛、文档、培训以及其他资源。



图1-23 AWS管理控制台导航栏



接下来，我们需要创建一个用于连接虚拟服务器的密钥对。

### 1.8.3 创建一个密钥对

要访问AWS中的虚拟服务器，客户需要一个由私钥和公钥组成的密钥对。公钥将上传到AWS并配置到虚拟服务器中，而私钥是客户私有的。这有点儿类似于密码，但更安全。一定要保护好自己的私钥，它就像是密码一样。这是私有的，所以不要弄丢它。一旦丢失，就无法重新获得。

要访问Linux服务器，请使用SSH协议。客户将在登录时通过密钥对而不是密码进行身份验证。如果客户需要通过远程桌面协议（RDP）来访问Windows服务器，客户也需要使用密钥对才能解密管理员密码，然后才能登录。

以下步骤将引导客户进入提供虚拟服务器的EC2服务的仪表板，在那里客户可以获取密钥对。

- （1）在<https://console.aws.amazon.com>上打开AWS管理控制台。
- （2）点击导航栏中的“服务”，找到“EC2服务”并点击它。
- （3）客户的浏览器将显示EC2管理控制台。

如图1-24所示，EC2管理控制台被分成3列。第一列是EC2导航栏，因为EC2是最早的服务，所以它具有许多可以通过导航栏可以访问的功能；第二列简要介绍了所有的EC2资源；第三列提供了附加信息。

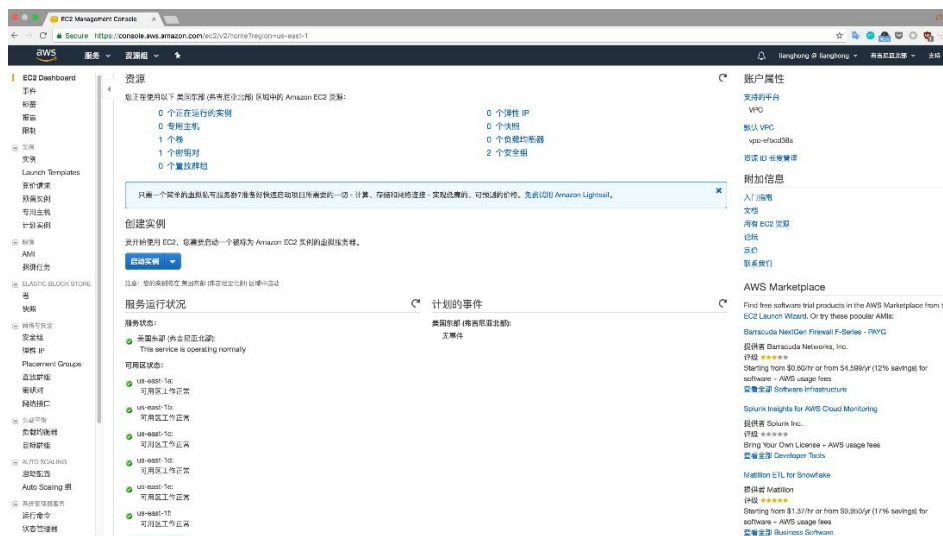


图1-24 EC2管理控制台

以下步骤可以用来创建一个新的密钥对。

(1) 在“网络与安全”下的导航栏中点击“密钥对”。

(2) 点击图1-25所示的页面上的“创建密钥对”按钮。

(3) 将密钥对命名为**mykey**。如果客户选择了其他名字，则必须在后续的所有示例中替换密钥对的名字！

在密钥对创建期间，客户要下载一个名为**mykey.pem**的文件。你现在必须准备好该密钥对以备将来使用。根据所使用的操作系统，你可能需要采取不同的操作。因此需要阅读对应的操作系统的部分。

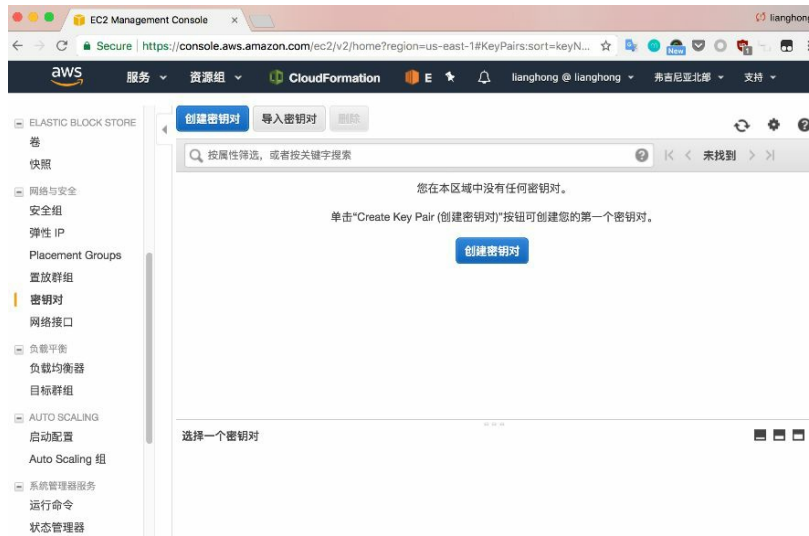


图1-25 EC2管理控制台上的密钥对

#### 使用自己的密钥对

读者也可以将现有的密钥的公钥上传到AWS。这样做有以下两个优点。

- 可以复用现有的密钥对。
- 可以确定只有自己知道密钥对的私钥部分，如果使用“创建密钥对”按钮，你可能担心AWS知道（至少暂时的）自己的私钥。

我们决定在本书中不采用这个做法，因为在一本书里面用这种做法会有一些不方便的地方。

## 1. Linux与Mac OS X

客户现在唯一需要做的是更改mykey.pem的访问权限，以便只有自己可以读取该文件。为此，请在终端中运行命令`chmod 400 mykey.pem`。当读者需要在本书中首次登录虚拟服务器时，将会了解如何使用密钥。

## 2. Windows

Windows不提供SSH客户端，因此客户需要下载适用于Windows的PuTTY安装程序，然后安装PuTTY。PuTTY提供了一个名为PuTTYgen

的工具，可以将mykey.pem文件转换为mykey.ppk文件，客户需要按照以下步骤操作。

- （1）需要运行应用程序PuTTYgen，界面如图1-26所示。
- （2）在“Type of key to generate”（要生成的密钥类型）下选择“SSH-2 RSA”。
- （3）点击“Load”（加载）。
- （4）因为PuTTYgen仅显示\*.ppk文件，需要将“文件名”字段的文件扩展名切换到“所有文件”。
- （5）选择mykey.pem文件，然后点击“Open”（打开）。
- （6）确认对话框。
- （7）将“Key comment”改成mykey。
- （8）点击“Save private key”。在没有密码的情况下，忽略关于保存密钥的警告。



图1-26 PuTTYgen允许将下载的pem文件转换成PuTTY需要的.pkk文件

.pem文件现在已转换为PuTTY所需的.pkk格式。当需要在本书中首次登录虚拟服务器时，我们会学习如何使用密钥。

## 1.8.4 创建计费告警

在开始使用AWS账户之前，我们建议读者创建一个计费告警。如果超过免费套餐的额度，读者会收到告警邮件。本书中的示例如果超出了免费套餐的范围会给出一个提醒。为了确保在清理过程中没有遗漏任何内容，读者要按照AWS建议创建一个计费告警。

## 1.9 小结

- Amazon Web Services (AWS) 是一个Web服务平台，为计算、存储和连网提供解决方案。
- 节约成本并非使用AWS的唯一好处，客户还将从灵活的容量、容错服务和全球基础设施的创新以及快速发展的平台中受益。
- 无论是应用广泛的Web应用，还是具有高级网络设置的专业的企业级应用，任何使用场景都可以在AWS上实现。
- 可以用许多不同的方式与AWS交互。可以使用基于Web的图形用户界面（GUI）来控制不同的服务，使用程序代码从命令行或者SDK中以编程的方式管理AWS，或者使用蓝图在AWS上设置、修改或删除AWS上的基础设施。
- 按使用付费是AWS服务的定价模式。计算能力、存储和网络服务的收费类似于电力。
- 创建一个AWS账户很容易。现在我们已经了解了如何设置密钥对，可以登录到虚拟服务器，供以后使用。

## 第2章 一个简单示例：5分钟搭建WordPress站点

### 本章主要内容

- 创建一个博客站点的基础设施架构
- 分析博客站点基础设施架构的成本
- 探索一个博客站点的基础设施架构
- 关闭博客站点的基础设施

在第1章中，我们了解了为什么AWS是运行Web应用的绝佳选择。在本章中，我们评估将一个博客站点的基础设施架构从假想公司的服务器迁移至AWS。

#### 示例都包含在免费套餐中

本章中的所有示例都包含在免费套餐中。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

你假想的公司目前在自己的服务器上使用WordPress来承载超过1000篇博客内容。由于用户不能忍受服务中断，博客站点的基础架构必须具备高可用的特点。为了评估这个迁移是否可行，需要完成如下工作。

- 建立一个具有高可用特性的博客站点的基础设施。
- 评估基础设施每月的成本。

WordPress用PHP编写，使用MySQL数据库存储数据，由Apache作为Web服务器来展现页面。根据这些信息，现在把你的需求映射到AWS服务之上。

## 2.1 创建基础设施

可以使用4种不同的AWS服务把旧的基础设施复制到AWS。

- 弹性负载均衡（Elastic Load Balancing, ELB）——AWS提供的弹性负载均衡服务。ELB将流量分发到它后面的一组服务器上，并且它自身默认就是高可用的。
- 弹性计算云（Elastic Compute Cloud, EC2）——EC2服务提供的虚拟服务器。你将使用一个Linux服务器来安装Apache、PHP和WordPress。这个例子选用的服务器的操作系统Amazon Linux，一个针对云计算优化过的Linux发行版。你也可以选择Ubuntu、Debian、Red Hat或者Windows等。因为虚拟服务器有可能会宕机，因此你需要部署至少两台，并通过ELB分发流量。一旦一台服务器宕机，ELB将会停止给宕机的服务器发送流量。在宕机的服务器被替换之前，余下的一台服务器将要承担全部的访问请求。
- 适用于MySQL的关系数据库服务（Relational Database Service for MySQL）——WordPress基于流行的MySQL数据库。而AWS的关系数据库服务（Relational Database Service, RDS）提供了对MySQL的支持。你在选择了数据库的规格（存储、CPU、RAM）以后，RDS会负责其余的工作（备份、升级）。并且，RDS也可以通过数据复制实现MySQL的高可用。
- 安全组（Security group）——安全组类似于防火墙，是AWS控制网络流量的一项基本服务。安全组这项服务能够附加到ELB、EC2、RDS等服务上。通过设置和附加安全组，ELB可以只接受对80端口访问的互联网流量，Web服务器只接受来自ELB对服务器80端口的访问请求，而MySQL数据库则只接受来自Web服务器对于3306端口的连接。如果想通过SSH直接登录Web服务器，那么需要打开22端口。

图2-1展示了需要部署的全部基础设施。看起来有不少工作要做，让我们开始吧！



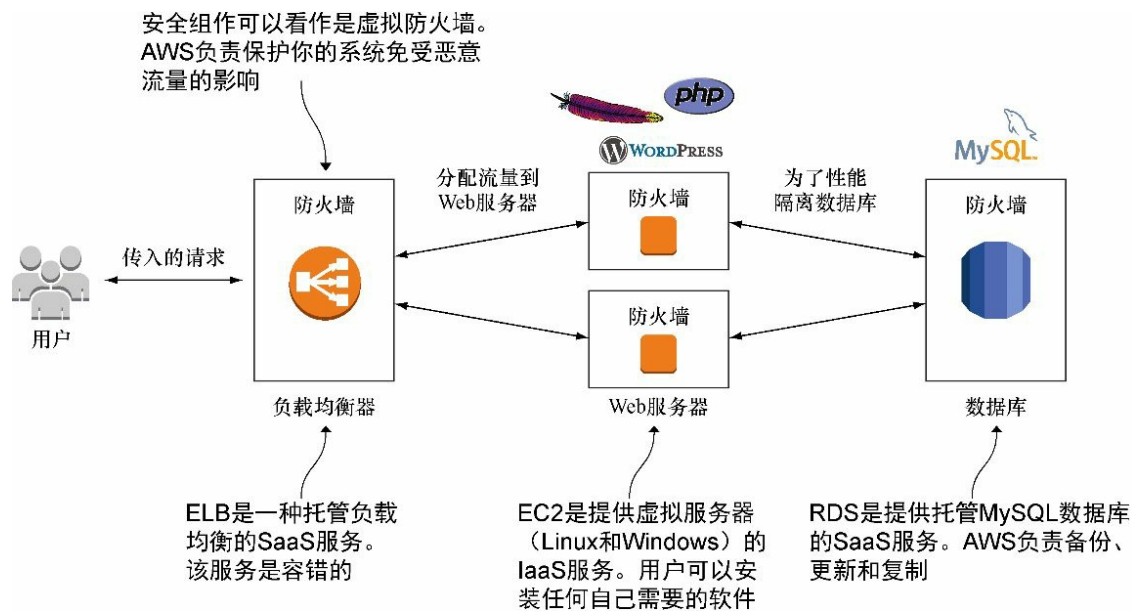


图2-1 该公司的博客网站的基础设施包括两台负载均衡的Web服务器 运行了WordPress和一台MySQL数据库服务器

如果你以为搭建步骤会有很多页，那么你现在可以高兴一下了。因为创建所有基础架构仅需在图形界面上完成一些点击操作，然后后台就会自动完成下列任务。

- (1) 创建一个ELB。
- (2) 创建一个应用MySQL数据库的RDS。
- (3) 创建并附加上安全组。
- (4) 创建两个Web服务器：
  - 创建两个EC2虚拟服务器；
  - 安装Apache和PHP，使用的安装命令为`yum install php, php-mysql, mysql, httpd`；
  - 下载并解压最新版本的WordPress；
  - 使用已创建的RDS MySQL数据库来配置WordPress；
  - 启动Apache Web服务器。

为了创建博客站点的基础设施，要打开AWS管理控制台并登录。点击导航栏中的“服务”，然后点击“CloudFormation”服务，将看到图2-2所示的界面。

点击从蓝图创建一个新的基础设施

重新加载页面

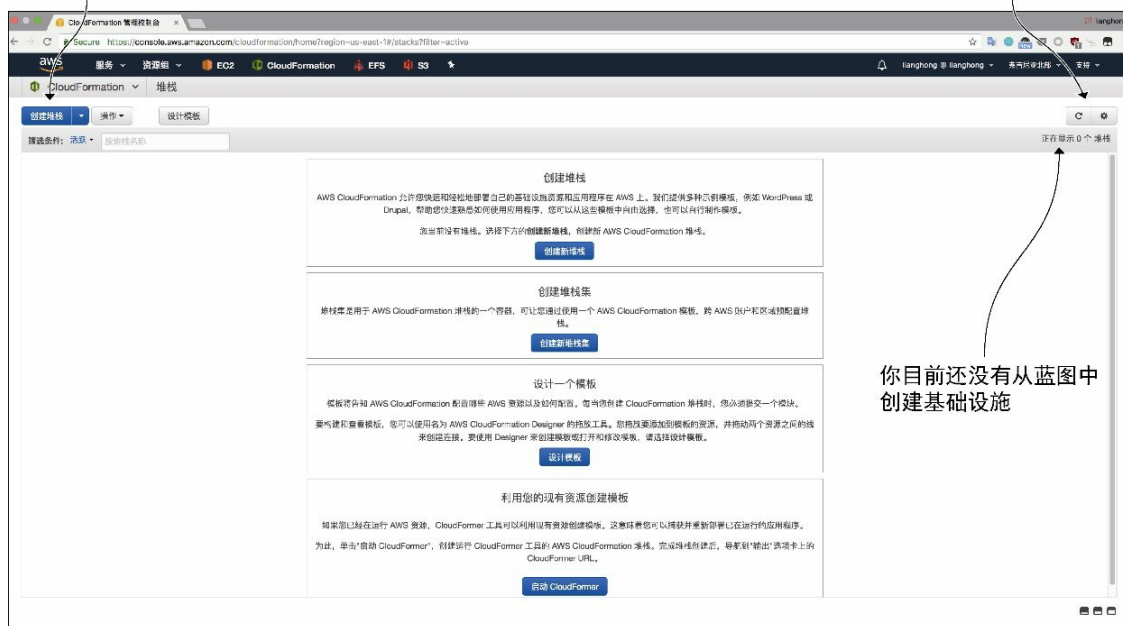


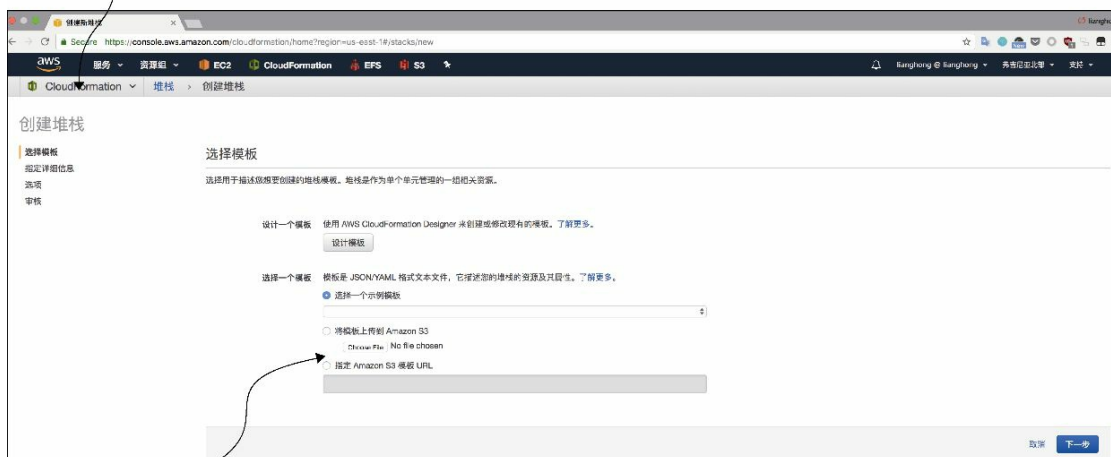
图2-2 CloudFormation屏幕界面

#### 注意

本书中的所有示例均使用弗吉尼亚北部（N.Virginia，也称为us-east-1）作为默认区域，如果没有额外的声明即使用该默认区域。在开始工作之前，要确保所选区域是弗吉尼亚北部。在AWS管理控制台的主导航栏右侧，可以确认或更换当前区域。

点击“创建堆栈”启动开始向导，共有4个步骤，如图2-3所示。

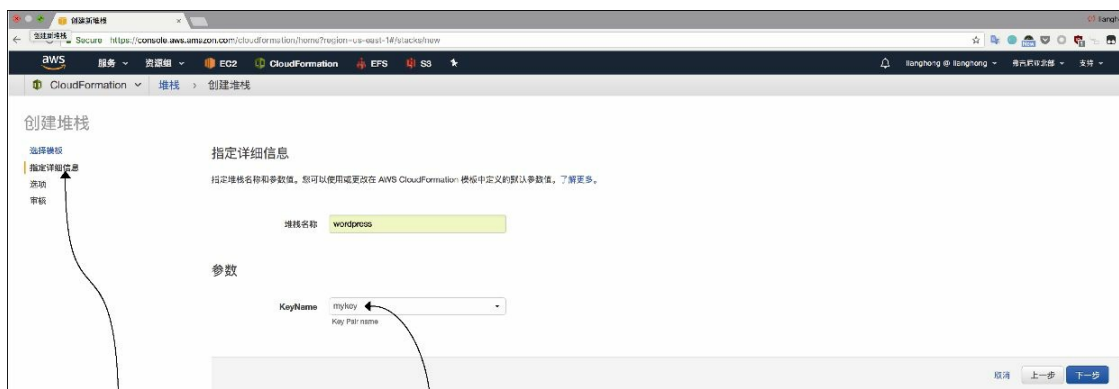
4步中的第1步



在这里可以选择基础设施的蓝图。可以选择一个样本、上传或提供URL。插入CloudFormation模板的URL

图2-3 创建一个博客站点的基础设施：第1步

在“选择一个模板”中选择“指定Amazon S3模板URL”，并且输入 <https://s3.amazonaws.com/awsinaction/chapter2/template.json>。然后点击“指定详细信息”，将堆栈名称设置为wordpress，将KeyName设置为mykey，如图2-4所示。



4步中的第2步

指定我们在第1章建立的键对

图2-4 创建一个博客站点的基础设施：第2步

点击“下一步”，为基础设施打上标签（tag）。标签是由一个键值对组成，并且可以添加到基础设施的所有组件上。通过使用标签，可以区分测试和生产资源，也可以添加部门名称以追踪各部门成本，还可以在一个AWS账号下运行多个应用时为应用标记所关联的资源。

## 媒体上传和插件

WordPress使用MySQL数据库存储文章和用户数据。但在默认设置下，WordPress把上传的媒体文件和插件存储在一个名为wp-content的本地文件目录下，所以服务器不是无状态的。如果要使用多台服务器，就需要每个请求可以被任何一个服务器处理，但是由于上传的媒体和插件只保存在某一台服务器上，所以默认配置并不支持多台服务器的部署方式。

因为没有解决上述问题，所以本章中的示例还不够完整。如果读者有兴趣了解进一步的解决方案，参见第14章。第14章将介绍如何在启用虚拟机时自动安装WordPress插件，以及上传的媒体文件是如何集中保存到对象存储中的。

在这个示例中，我们将使用标签来标记WordPress系统的资源，这将有助于你以后轻松地找到自己的基础架构。使用**system**作为键，**wordpress**作为值。图2-5展示了标签的配置方法。

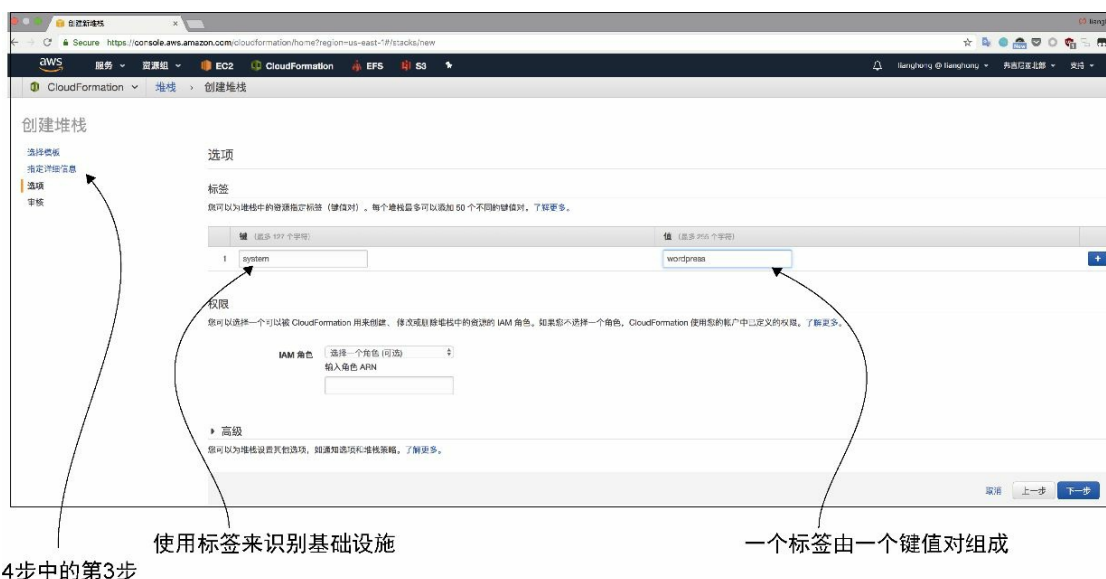


图2-5 创建一个博客站点的基础设施：第3步

点击“审核”，最后将看到一个确认页面，如图2-6所示。在“估算费用”一栏中，点击“费用”将在后台打开一个新标签页，我们将在下一节处理其中的内容。切换至原先的浏览器标签页，点击“创建”。

基础架构现在将被创建。如图2-7所示，名为wordpress的堆栈正处于CREATE\_IN\_PROGRESS状态。现在可以休息5~15 min，回来之后就会有惊喜。

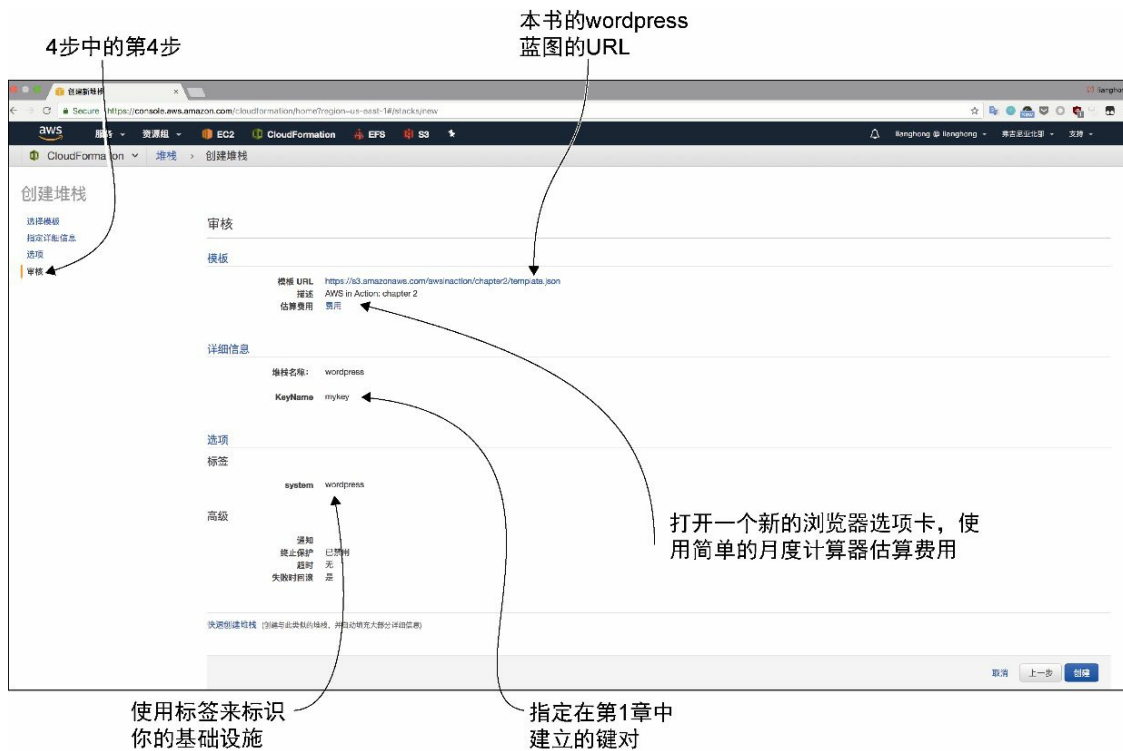


图2-6 创建一个博客站点的基础设施：第4步

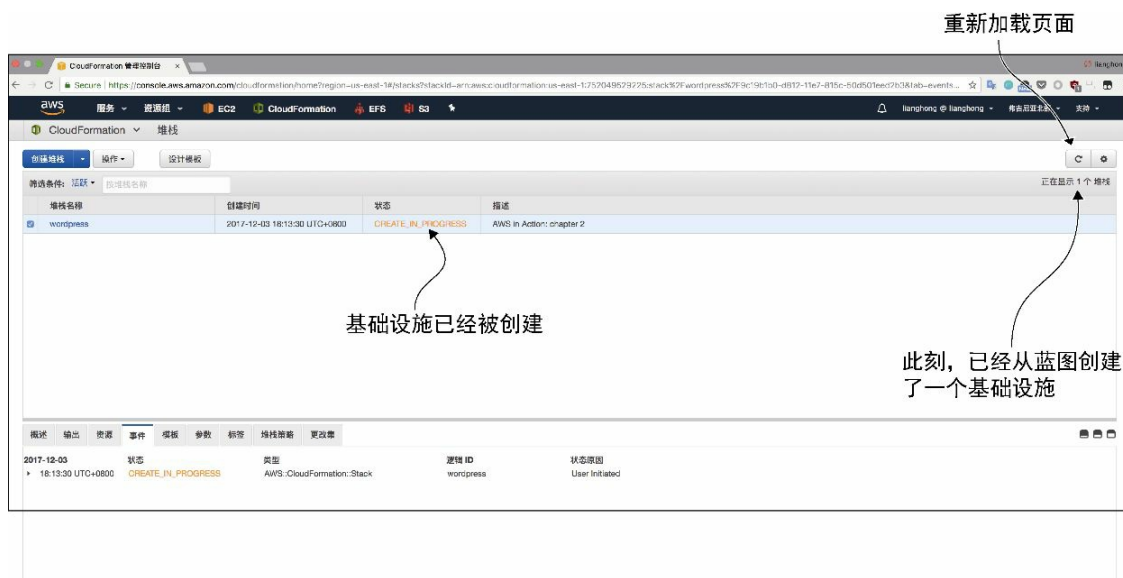


图2-7 审核界面

刷新页面查看结果，选择wordpress 一行，其状态应该是CREATE\_COMPLETE。如果状态仍然是CREATE\_IN\_PROGRESS，请耐心等待直到状态变为CREATE\_COMPLETE。如图2-8所示，切换到“输出”标签，将看到wordpress 系统的URL访问链接，点击该链接即可访

问。

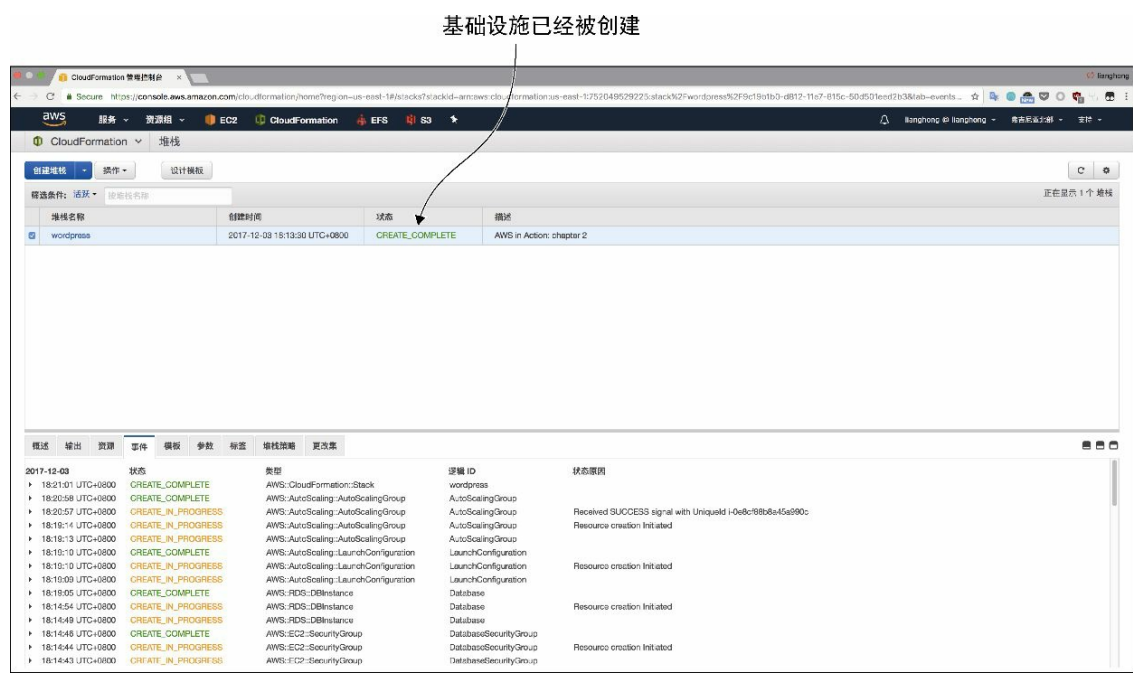


图2-8 博客站点基础设施的成果

看到这里读者可能会问：其工作原理是什么呢？答案就是自动化。

自动化参考

AWS的关键概念之一就是自动化。用户可以自动化一切AWS的服务。在后台，这个博客站点的基础设施是按照一个蓝图创建的。第4章将介绍更多关于这个蓝图的内容，以及针对基础设施的编程理念。第5章将介绍如何自动化安装软件。

下一节，我们将探索这个博客站点的基础设施，以便更好地了解正在使用的各种服务。

## 2.2 探索基础设施

现在我们已经创建了博客站点的基础设施，那就让我们一起来深入了解一下。基础设施包含了如下几个部分：

- Web服务器；
- 负载均衡器；
- MySQL数据库。

我们将使用控制台的资源组功能来总览所有内容。

### 2.2.1 资源组

资源组（**resource group**）是一个AWS资源的集合。资源是AWS服务或功能的抽象概念；资源可以是一台EC2服务器、一个安全组或者一个RDS数据库。资源可以用键值对作为标签来标记，而资源组可以指定拥有哪些标签的资源才能属于该组。此外，资源组会指定资源所处的区域（**region**）。当在一个AWS账户下运行多个系统时，客户可以使用资源组来归类各种资源。

还记得，我们之前给博客网站点基础设施标记的标签是，**system** 为键，**wordpress** 为值。在后文中，我们将采用（**system:wordpress**）这样的记法来表示键值对。这里将使用此标签来为WordPress的基础设施创建一个资源组。如图2-9所示，点击导航栏的“资源组”部分，然后点击“创建资源组”。



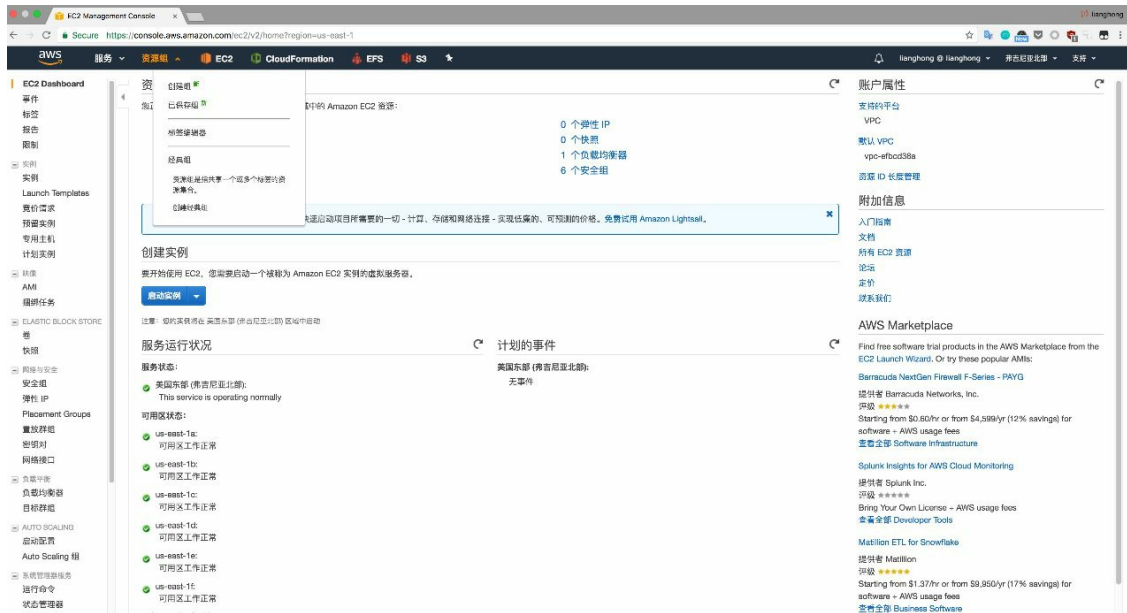


图2-9 建立一个新的资源组

#### 关于图中圆圈中的数字

在一些图中，如图2-9所示，我们将看到一些带圆圈的数字，它们标记了我们应该遵循的点击顺序，以便执行周围文字所述的流程。

现在我们将创建一个新的资源组。

- (1) 资源组的名字为**wordpress**，或者选择一个自己喜欢的名字。
- (2) 添加标签，键为**system**，值为**wordpress**。
- (3) 选择区域为弗吉尼亚北部。

填写的表单看起来如图2-10所示。现在，保存资源组。





图2-10 为博客站点建立一个资源组

## 2.2.2 Web服务器

现在我们将看到图2-11所示的界面，在左边栏的EC2分类下选择“实例”就可以看到Web服务器。点击“Go”列的箭头图标，可以很容易地查看某一个Web服务器的细节。

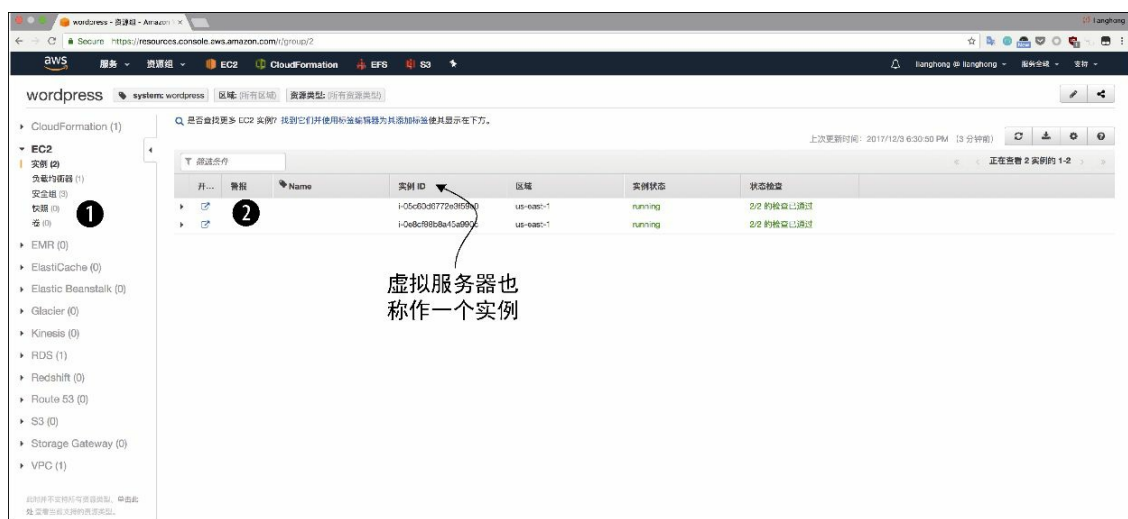
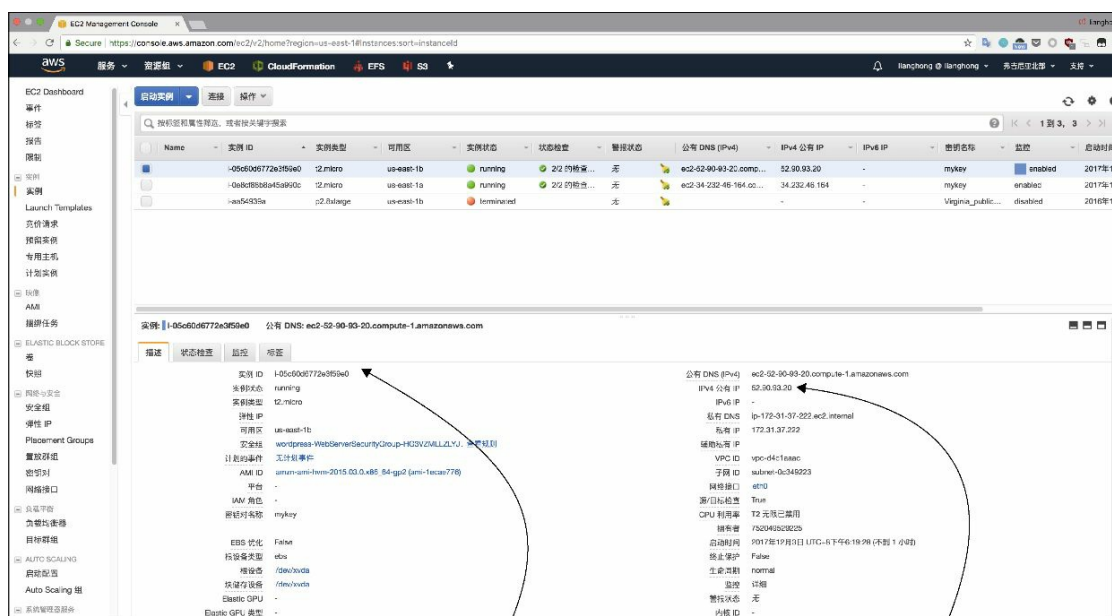


图2-11 在资源组中的博客站点Web服务器

我们现在看到的是Web服务器（也称为EC2实例）的细节。图2-12展示了将会看到的主要内容，一些有趣的细节如下。

- 实例类型 ——展示实例的处理能力。我们将在第3章了解到关于实例类型的更多的内容。
- 公有IP ——在互联网上可以访问的IP地址。我们可以使用SSH通过这个IP地址登录到服务器。
- 安全组 ——如果点击查看规则，将看到正在生效的防火墙规则。例如，允许所有的来源（0.0.0.0/0）访问端口22。
- AMI ID ——记住你正在使用的是Amazon Linux操作系统。如果点击AMI ID，将看到操作系统的版本等。



选择这个标签页可以  
查看一些监控图表

使用SSH连接到  
这个IP地址

图2-12 博客站点基础设施中Web服务器的细节信息

选择“监控”标签查看Web服务器的使用程度。这将成为你日常工作的一部分：掌握基础设施的实际运行情况。AWS收集了一些系统指标，并把它们展示在监控功能里面。如果CPU利用率高于80%，应该添加第三台服务器，以防止Web页面加载时间过长。

## 2.2.3 负载均衡器

点击左边栏EC2分类下的“负载均衡器”，可以看到所创建的负载均衡器，如图2-13所示。点击“开...”列中的箭头，就可以看到负载均衡器

的细节。



图2-13 博客站点基础设施资源组中的负载均衡器

现在看一下负载均衡器的细节。图2-14展示了将会看到的主要内容。最有趣的是，负载均衡器是如何将流量转发到Web服务器的。

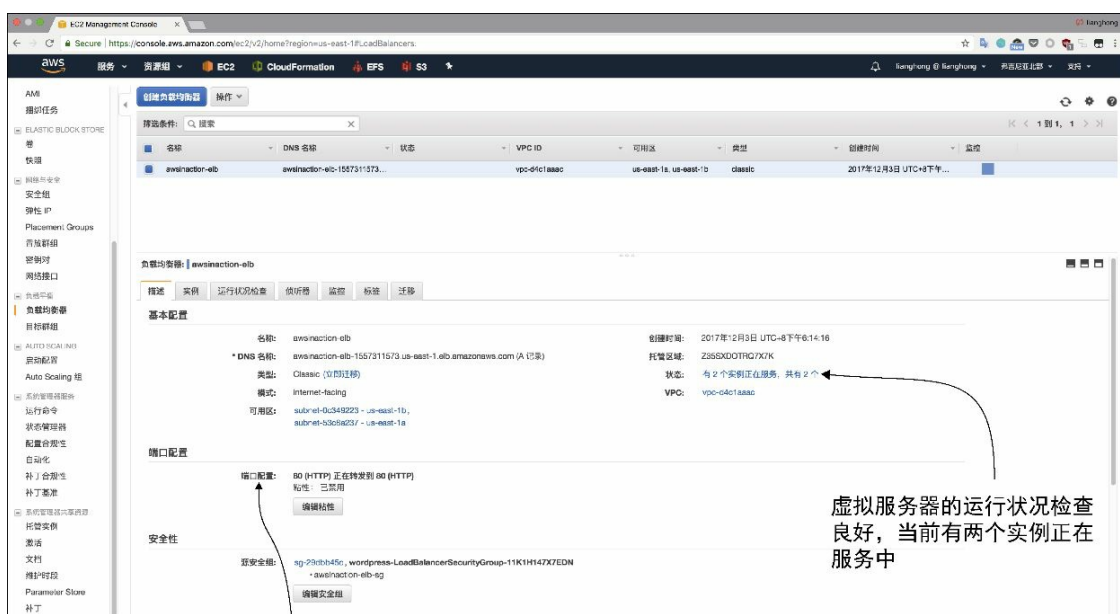


图2-14 负载均衡器服务于博客站点基础设施的细节

博客站点的基础设施的运行在80端口上，即HTTP协议的默认端口。创建的负载均衡器仅接受HTTP协议的连接，并会把一个请求转发

给后端的一台监听80端口的Web服务器上。负载均衡器还会对关联的虚拟服务器进行运行状况检查。因为两台虚拟服务器工作正常，所以负载均衡器就会将流量转发过去。

如前所述，“监控”标签页里包含了一些有趣的指标，我们应该在生产环境里予以关注。如果流量模型突然变化，那么系统可能出现了问题。显示出来的HTTP错误数的指标会帮助我们对系统进行监控和排错。

## 2.2.4 MySQL数据库

最后但并非不重要，我们一起来看一下MySQL数据库。在wordpress资源组中可以看到MySQL数据库。在左边栏选择RDS分类下的“数据库实例”，点击“开...”列的箭头（如图2-15所示），将看到数据库的细节信息。

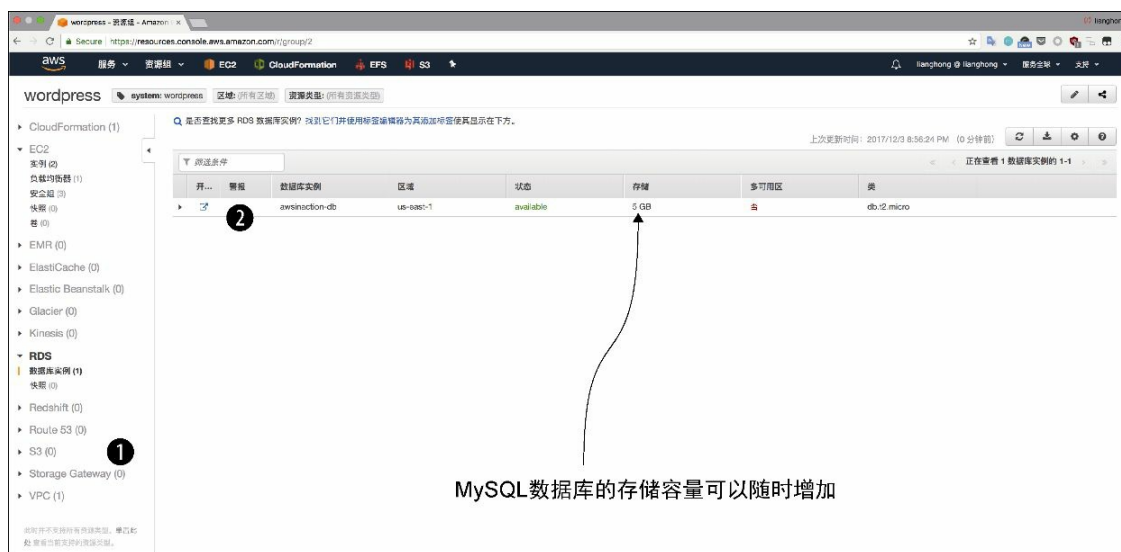


图2-15 博客站点基础设施资源组中的MySQL

图2-16展示了MySQL数据库的细节信息。使用RDS的好处是，因为AWS平台会自动完成数据库备份，客户不再需要关心这些工作。同时，在自定义维护时间窗口后，AWS也将自动完成数据库更新。记住，可以按照实际需要选择适当的数据库的存储、CPU和内存大小。AWS提供了许多不同的实例类型，从1个CPU核、1 GB内存，到32个

CPU核、244 GB内存，我们将在第9章了解到更多相关内容。

接下来，应该评估成本了，我们将在下一节分析博客站点基础架构的各项成本。



图2-16 博客站点基础设施中MySQL数据库的信息





Amazon RDS DB实例	MySQL数据库	12.45
Amazon RDS存储	MySQL数据库	0.58
总 计		57.37

请注意，这只是一个估算的成本。每个月底，你将收到根据实际使用计算出来的账单。所有资源按需使用，账单是按小时或按GB数量计算的。那么，哪些因素会影响基础设施的实际使用呢？

- 负载均衡器的流量 —— 因为通常人们会在12月和夏季去度假，不会去浏览博客，所以这个时间段的预期成本会下降。
- 数据库的存储 —— 如果你公司的博客文章在不断地增加，数据库的存储也会增加，从而导致数据库存储成本的增加。
- Web服务器的数量 —— 每台Web服务器都是按小时计费的。如果两台Web服务器不足以处理每天的全部流量，你可能需要部署第三台服务器，这就要消耗更多的虚拟服务器的运行小时数。

预估基础设施的成本是一项复杂的任务。即时没有在AWS上运行也是如此。此外，灵活性是使用AWS可以得到的好处之一。如果预估的Web服务器数量过多，可以随时停止一台或几台，这就意味着同时停止了计费。

现在你已经对博客站点基础设施的成本有了大概的了解，接下来我们该关闭基础设施并完成迁移评估。

## 2.4 删除基础设施

你成功地判断出了自己的公司可以将博客站点的基础设施迁移至AWS，并且每月的花费大约是60美元。现在你可以决定是否要继续执行这个迁移了。

为完成迁移评估，你需要删除博客站点基础设施使用的全部资源。因为使用并非真实的数据进行评估，所以不必担心数据丢失。

进入管理控制台的CloudFormation服务，执行如下操作。

- (1) 选择wordpress 这一行。
- (2) 点击“删除堆栈”，如图2-18所示。

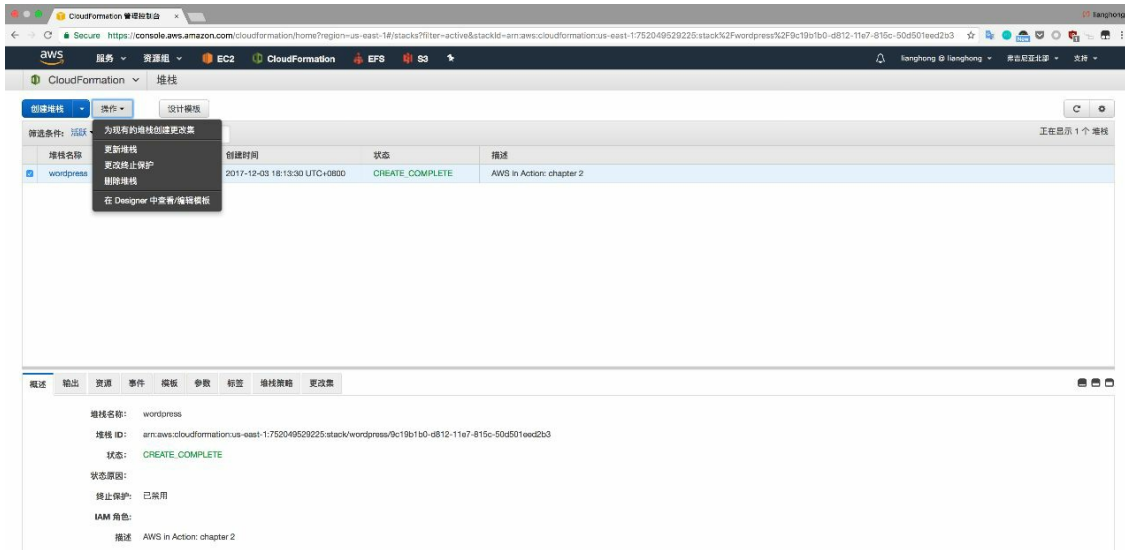


图2-18 删除博客站点的基础设施

如图2-19所示，在确认删除之后，AWS会在几分钟内自动分析资源依赖关系并删除整个基础设施。





图2-19 确认删除博客站点的基础设施

这是管理基础设施的一种高效方法。正如自动化创建基础设施一样，删除也是完全自动化的。你可以随时按需创建或删除基础设施，仅需在基础设施上创建和运行时为之支付费用。

## 2.5 小结

- 创建博客站点基础设施的工作是可以完全自动化的。
- 基础设施可以随时按需创建，不需要承诺使用时长。
- 用户需要按照使用基础设施小时数付费。
- 基础设施由多个组件（如虚拟服务器、负载均衡器和数据库）构成。
- 基础设施可以一键删除，处理流程是自动的。

## 第二部分 搭建包含服务器和网络的虚拟基础设施

计算能力和网络连接能力已经成为从小型个体、中型企业到大型集团的基本要求了。过去满足这类需求的方法是，在自有机房或者外包的数据中心维护这些硬件。如今，云计算提供了革命性的方法来获取计算能力。用户可以在需要的时候，几分钟内开启或停止虚拟服务器来满足对计算资源的需求。我们同样可以在虚拟服务器上自由安装软件，这使我们能够执行我们的计算任务，而不必购买或租用硬件设备。

如果想了解AWS，最好能够深入了解在表面功能之下提供支撑的API（应用编程接口）所能带来的各种可能性。用户可以利用向REST API发送请求控制所有AWS服务。在这些API上，我们可以搭建各种解决方案来实现基础设施自动化。基础设施自动化是云计算胜过自有设施的一个重要优势。

本书的这个部分的主题是调度基础设施并自动化部署应用。建立虚拟网络可以帮助你搭建封闭、安全的网络环境，并让它和你家里的网络或者企业内网实现互通。第3章探讨虚拟服务器，读者会了解EC2的核心概念。第4章讨论自动化基础设施甚至像管理代码一样管理它。第5章将展示3种不同的方法来想AWS部署软件。第6章是关于网络的，读者可以学到如何利用虚拟专用网和防火墙来保护自己的系统。

## 第3章 使用虚拟服务器：EC2

### 本章主要内容

- 启动一台Linux虚拟服务器
- 使用SSH远程连接到虚拟服务器
- 在虚拟服务器上监控和调试
- 减少在虚拟服务器上的开销

我们口袋中的智能手机和背包里的笔记本电脑可以达到惊人的计算能力。不过，如果我们需要大规模的计算能力和很高的网络流量，或者需要全天候不间断可靠运行，虚拟服务器则是更合适的选择。有了虚拟服务器，就拥有了数据中心里一台物理服务器的一部分。在AWS中，有一个叫弹性计算云（Elastic Compute Cloud，EC2）的服务用来提供虚拟服务器。

## 3.1 探索虚拟服务器

虚拟服务器是一台物理服务器的一部分。物理服务器通过软件来隔离其上的各个虚拟服务器。一台虚拟服务器由CPU、内存、网络接口和存储组成。物理服务器也称为宿主服务器（**host server**），其上运行的虚拟服务器称为客户机（**guest**）。虚拟化管理器（**hypervisor**）负责孤立各个客户机并调度它们对硬件的请求。图3-1展示了服务器虚拟化的各个层次。

不是所有示例都包含在免费套餐中

本章中的示例不都包含在免费套餐中。当一个示例会产生费用时，会显示一个特殊的警告消息。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

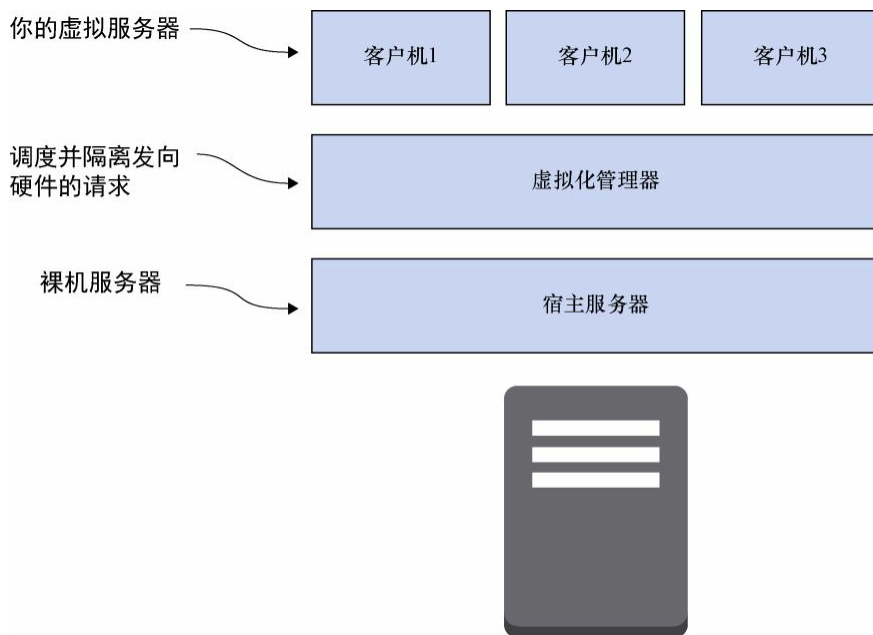


图3-1 服务器虚拟化的层次关系

下面是一些典型的虚拟服务器应用案例：

- 网络应用服务器主机；

- 运行企业内应用；
- 数据转换或分析。

### 3.1.1 启动虚拟服务器

要启动一台虚拟服务器只需要以下几次简单的点击。

(1) 打开AWS管理控制台。

(2) 确保在“美国东部（弗吉尼亚北部）”区域（见图3-2），我们的示例专门为此区域进行了优化。



图3-2 确保在正确的区域

(3) 在导航栏中展开“服务”列表，找到EC2服务并打开，会看到一个图3-3所示的页面。



图3-3 EC2服务的界面和“启动实例”按钮

(4) 点击“启动实例”按钮来执行启动虚拟服务器向导。

这个向导将带用户经历以下几个步骤。

(1) 选择操作系统。

(2) 选择虚拟服务器的规格。

(3) 配置详细信息。

(4) 检查输入并为SSH选择一个密钥对。

## 1. 选择操作系统

第一步是为虚拟服务器选择操作系统和预安装软件的组合，我们称其为Amazon系统映像（Amazon Machine Image, AMI）。为虚拟服务器选择Ubuntu Server 14.04 LTS（HVM），如图3-4所示。

虚拟服务器是基于AMI启动的。AMI由AWS、由第三方供应商及社区提供。AWS提供Amazon Linux AMI，包含了为EC2优化过的从Red Hat Enterprise Linux派生的版本。用户也可以找到流行的Linux版本及Microsoft Windows Server的AMI。另外，AWS Marketplace提供预装了第三方软件的AMI。

### AWS上的虚拟设备

一台虚拟设备（virtual appliance）是一个包含操作系统和预安装软件的映像，它可以运行在虚拟机管理程序（hypervisor）上。虚拟机管理程序的工作就是运行一台或多台虚拟设备。因为一台虚拟设备包含一个固定状态，每次启动这台虚拟设备，都会得到完全一致的结果。用户可以经常根据自己的需要再生产虚拟设备，这样就能利用它们来消除安装配置复杂软件的开销。虚拟设备可以被常见的虚拟化工具使用，以基础设施即服务的方式在云中提供。这些虚拟化工具可能来自于VMware、Microsoft或Oracle等厂商。

AMI是AWS上的虚拟设备映像。它是一个特殊的虚拟设备，用于EC2服务商的虚拟服务器。从技术上来说，AMI由一个只读文件系统，包括操作系统、额外的软件和配置构成；它不包含操作系统内核。操作系统内核从Amazon Kernel Image（AKI）装载。也可以利用AMI在AWS上部署软件。

AWS使用Xen，一个开源的虚拟机管理程序，作为EC2服务的底层技术。AWS上这一代的虚拟服务器使用硬件辅助的虚拟化技术。这个技术被称为Hardware Virtual Machine（HVM）且使用Intel VT-x平台。一台运行在基于HVM的AMI的虚拟服务器使用完全

虚拟化的硬件设备，并且可以利用硬件设备扩展，它提供快速访问底层硬件。

为虚拟Linux服务器使用3.8+内核将提供最好的性能。要这样做，用户应该使用至少Amazon Linux 13.09、Ubuntu 16.x或RHEL 7。如果要启动新的虚拟服务器，一定要确保自己使用的是HVM映像。

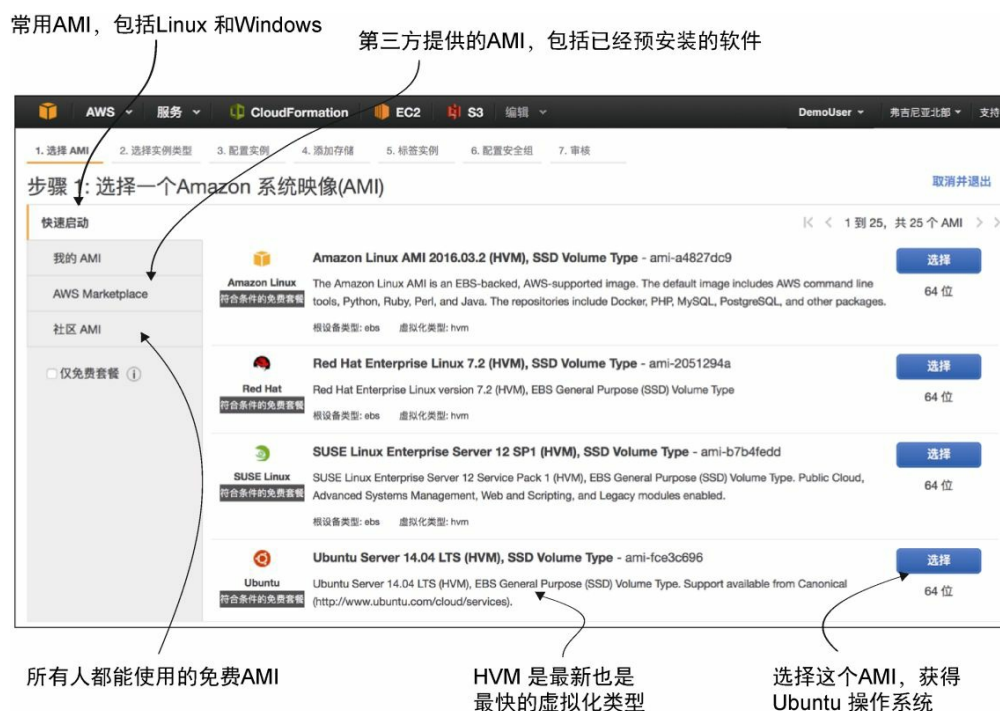


图3-4 为虚拟服务器选择操作系统

## 2. 选择虚拟服务器的尺寸

现在是时候来为虚拟服务器选择所需的计算能力了。图3-5展示了向导的下一步。在AWS上，计算能力被归类到实例类型中。一个实例类型主要描述了CPU的个数及内存数量等资源。





图3-5 选择虚拟服务器的尺寸

#### 实例类型及家族

不同实例类型用同样的结构化方式命名。实例家族（instance family）用相同的方式对实例类型进行分组。AWS不时发布新的实例类型及家族；不同版本的硬件用“代”（generation）来表示。实例尺寸（instance size）定义了CPU的处理能力、内存、存储及网络。

例如，实例类型t2.micro告诉用户以下信息。

- （1）实例家族是t。它包含了小的、便宜的虚拟服务器，具备最低基线的CPU性能，但是有能力突然在短时间内大大超过其CPU性能基线。
- （2）用户正在使用这一实例类型的第二代。
- （3）尺寸是micro，意味着这个实例非常小。

表3-1展示了不同案例下使用的实例类型的示例。所有的价格（以美元为单位）对美国东部（弗吉尼亚）有效，且虚拟服务器是基于2015年4月14日的Linux版本的。

表3-1 实例家族及实例类型概览

实例类	虚拟	内存	描 述	典型案例	小时价格
-----	----	----	-----	------	------

型	CPU (个)	(GB)			(美元)
t2.micro	1	1	最小及最便宜的实例家族，一般性能基线，有能力短时间突破CPU性能基线	测试与开发环境，以及低流量的应用	0.013
m3.large	2	7.5	有平衡比例的CPU、内存；还有一定的网络性能	所有类型的应用，如中型数据库、HTTP服务器，以及企业级应用	0.140
r3.large	2	15	使用额外内存为内存密集型应用做了优化	内存中缓存，企业级应用服务器	0.175

另外，还有为计算密集型工作量、高网络I/O型工作量和存储密集型工作量做了优化的实例类型与家族。还有实例类型为服务器端图形化工作量提供GPU访问。我们的经验表明，用户会高估自己的应用所需的资源，因而我们推荐读者先尝试使用比自己首先想到的要小一些的实例类型来启动自己的应用。

计算机的运算速度越来越快，而且技术越来越专业化。AWS持续不断引入新的实例类型与家族。它们中有些是对已存在的实例家族的改进，而另一些则专注于特殊的工作负载。例如，实例家族d2于2015年3月被引入。它提供了为要求高顺序读写访问工作量优化的实例，如一些数据库及日志处理。

用户进行最初的实验时使用最小且最便宜的虚拟服务器就足够了。在图3-6所示的向导界面上，选择实例类型t2.micro，然后点击“下一步：配置实例详细信息”按钮来继续。

### 3. 实例详细信息、存储、防火墙和标签

向导的接下来4个步骤十分容易，因为不需要更改默认值。我们稍后会学习这些设置的详细信息。

图3-6展示了向导的下一步。用户可以更改自己的虚拟服务器的详细信息，如网络配置或需要启动的服务器的数量。目前，保持默认值，然后点击“下一步：添加存储”按钮。

有不同的在AWS上存储数据的选项，我们将在后面的章节进行介绍。图3-7展示了向虚拟服务器添加网络附加存储的选项。保持默认值，然后点击“下一步：添加标签”按钮。

清晰的组织分类可以营造整洁的环境。在AWS平台上，使用标签可以帮助用户很好地组织资源。标签是一个键值对。用户至少应该给自己的资源添加一个名称标签，以便今后更方便地找到它。使用**Name** 作为键，**myserver** 作为值，如图3-8所示。然后点击“下一步：配置安全组”按钮。



图3-6 虚拟服务器的详细信息



图3-7 为虚拟服务器添加网络附加存储



图3-8 用一个名称标签为虚拟服务器命名

防火墙帮助用户保护虚拟服务器的安全。图3-9展示了一个防火墙设置，该设置允许从任意位置使用SSH访问默认的22端口。这正是我们想实现的效果，所以保留默认值，然后点击“审核和启动”按钮。



图3-9 为虚拟服务器配防火墙

## 4. 检查输入并为SSH选择一个密钥对

启动虚拟服务器的步骤快要完成了。向导会向用户显示新的虚拟服务器的总结信息（见图3-10）。确保选择了Ubuntu Server 16.04 LTS（HVM）作为操作系统，实例类型为t2.micro。如果一切正确，点击“启动”按钮。

在这一案例中需要允许从任意地方访问SSH，因此这里产生一个警告



图3-10 检查虚拟服务器启动的实例

最后但同样重要的是，向导要求用户提供新虚拟服务器的密钥对。

#### 丢失自己的密钥？

用户需要一个密钥来登录到自己的虚拟服务器。用户使用一个密钥而不是密码来完成身份认证。密钥比密码更加安全，而且在AWS上运行Linux的虚拟服务器强制SSH访问使用密钥方式。如果跳过了1.8.3节中创建密钥的步骤，可以根据下面的步骤来创建一个个人密钥。

- (1) 打开AWS管理控制台。在导航栏的“服务”的下面找到“EC2服务”，然后点击它。
- (2) 切换到子菜单“密钥对”。
- (3) 点击“创建密钥对”。
- (4) 输入mykey 作为密钥对名字，然后点击“创建”，浏览器将自动下载密钥。
- (5) 打开一个终端，切换到下载目录。

(6) 仅限OS X和Linux：在控制台运行 `chmod 400 mykey.pem` 来修改访问文件 mykey.pem 的权限。

仅限Windows：Windows没有自带SSH客户端，所以需要安装PuTTY。PuTTY带有一个工具叫作PuTTYgen，它可以将mykey.pem文件转换成将需要的mykey.ppk。打开PuTTYgen，然后在“Type of key to generate”中选择“SSH-2 RSA”。点击Load。因为PuTTYgen只显示\*.ppk文件，需要切换“File name input”框中的文件扩展名为“所有文件”。现在可以选择mykey.pem文件，然后点击Open，确认对话框。修改“Key Comment”为mykey，然后点击“Save private key”。忽略关于未使用密码保护保存的密钥的警告。.pem文件现在被转换成了PuTTY所需要的.ppk格式。



读者可以在第1章找到关于如何创建密钥的详细说明。

选择“选择现有密钥对”选项，选择密钥对mykey，然后点击“启动实例”按钮（见图3-11）。

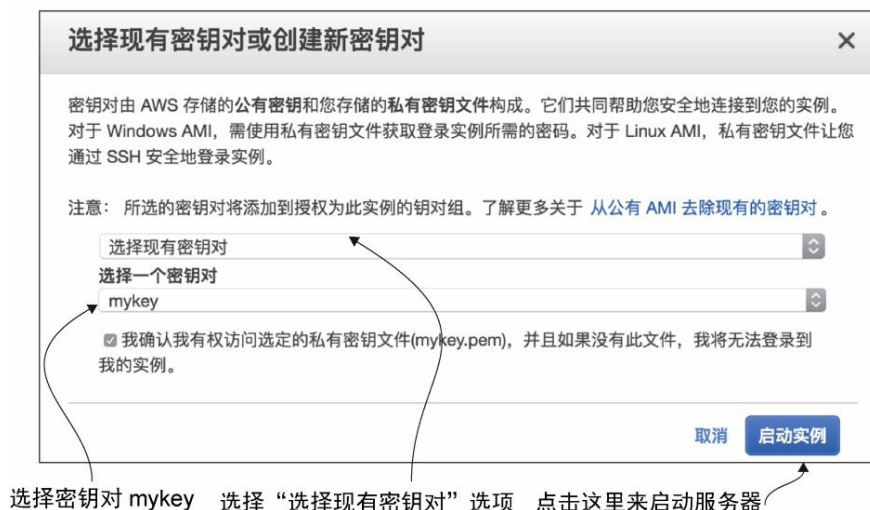


图3-11 为虚拟服务器选择一个密钥对

虚拟服务器启动了。点击“查看实例”来打开概览，然后等待虚拟服务器变为Running状态。要完全控制自己的虚拟服务器，用户需要远程登录。

### 3.1.2 连接到虚拟服务器

用户可以远程在虚拟服务器上安装额外的软件及运行命令。要登录到虚拟服务器，用户要找到服务器的公有IP地址。

（1）在导航栏下的“服务”中点击EC2，然后在左边子菜单中点击“实例”跳转到虚拟服务器的概览页。

（2）在表格中选择虚拟服务器。图3-12展示了服务器概览以及可以进行的操作。

（3）点击“连接”按钮，打开连接到虚拟服务器的说明。

(4) 图3-13展示了连接到虚拟服务器的对话框。找到虚拟服务器的公有IP地址，例如，在这个示例中为52.205.115.207。

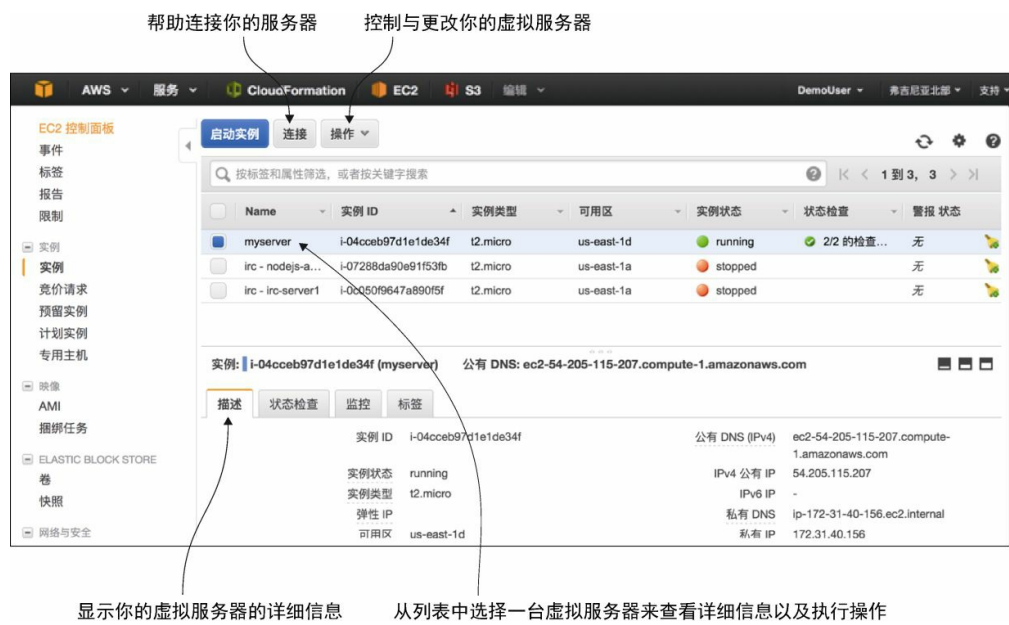


图3-12 虚拟服务器概览及操作控制



图3-13 使用SSH连接虚拟服务器的说明



有了公有IP地址及用户的密钥，用户就能够登录虚拟服务器了。下一节将根据我们的操作系统及本地机器来继续讲解。

## 1. Linux和Mac OS X

打开终端，输入`ssh -i $PathToKey/mykey.pem ubuntu@$PublicIp`，使用在1.8.3节中下载的密钥文件的路径替换`$PathToKey`，使用在AWS管理控制台的连接对话框中显示的公有IP地址替换`$PublicIp`。在关于新主机的认证的安全警告处回答Yes。

## 2. Windows

按下列步骤进行操作。

（1）找到在1.8.3节中创建的mykey.ppk文件，然后双击打开它。

（2）PuTTY Pageant应该会在任务条中显示为一个图标。如果没有，可能需要按照1.8.3节中的描述安装或重新安装PuTTY。

（3）启动PuTTY。填写AWS管理控制台的连接对话框中显示的公有IP地址，然后点击“Open”（见图3-14）。

（4）在关于新主机的认证的安全警告处回答Yes，然后输入ubuntu作为登录名，按Enter键。



图3-14 在Windows上使用PuTTY连接虚拟服务器

### 3. 登录信息

不论使用的是Linux、Mac OS X还是Windows，当登录成功后用户都应该会看见如下信息：

```
ssh -i ~/Downloads/mykey.pem ubuntu@52.4.216.201
Warning: Permanently added '52.4.216.201' (RSA) to the list of known hosts
.
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-44-generic x86_64)

* Documentation: https://help.ubuntu.com/

System information as of Wed Mar 4 07:05:42 UTC 2015

System load: 0.24           Memory usage: 5% Processes:      83
Usage of /:  9.8% of 7.74GB Swap usage:   0% Users logged in: 0

Graph this data and manage this system at:
https://landscape.canonical.com/
```

```
Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud
```

```
0 packages can be updated.
0 updates are security updates.
```

```
The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
```

```
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

```
~$
```

现在我们已经连接上了虚拟服务器，为运行命令做好了准备。

### 3.1.3 手动安装和运行软件

我们已经启动了一台Ubuntu操作系统的虚拟服务器。在程序包管理软件**apt**的帮助下，我们很容易安装额外的软件。作为开始，我们将安装一个叫**linkchecker**的小工具，它能让我们找到网站上断裂的链接：

```
$ sudo apt-get install linkchecker -y
```

现在就可以检查那些指向已经不存在的网站的超链接了。先选择一个网站，然后运行下面的命令：

```
$ linkchecker https://...
```

链接检查的结果看上去像下面这样显示：

```
[...]
URL      'http://www.linux-mag.com/blogs/fablesen'
Name     'Frank Ableson's Blog'
Parent URL http://manning.com/about/blogs.html, line 92, col 27
Real URL  http://www.linux-mag.com/blogs/fablesen
Check time 1.327 seconds
Modified  2015-07-22 09:49:39.000000Z
Result    Error: 404 Not Found

URL      '/catalog/dotnet'
Name     'Microsoft & .NET'
Parent URL http://manning.com/wittig/, line 29, col 2
Real URL  http://manning.com/catalog/dotnet/
Check time 0.163 seconds
D/L time  0.146 seconds
Size      37.55KB
Info      Redirected to 'http://manning.com/catalog/dotnet/'.
          235 URLs parsed.
Modified  2015-07-22 01:16:35.000000Z
Warning   [http-moved-permanent] HTTP 301 (moved permanent)
          encountered: you should update this link.
Result    Valid: 200 OK
[...]
```

根据网页数量的不同，网页爬虫需要一些时间来检查所有的网页是否有断裂的链接。最终它会列出所有断裂的链接，给用户机会找到并修复它们。

## 3.2 监控和调试虚拟服务器

如果用户需要找到应用出错或异常的原因，使用工具来帮助监控和调试就很重要了。AWS提供了工具来让用户监控和调试自己的虚拟服务器。其中有一种方法是检查虚拟服务器的日志。

### 3.2.1 显示虚拟服务器的日志

假如用户需要找出自己的虚拟服务器在启动时及启动后做了些什么，有一个简单的解决方案。AWS允许用户使用管理控制台（就是用来启动和关闭虚拟服务器的网络交互界面）显示服务器的日志。用户可按下面的步骤打开虚拟服务器日志。

- （1）在主导航栏中打开EC2，然后从子菜单中选择“实例”。
- （2）在表中点击一行，以选择正在运行的虚拟服务器。
- （3）在“操作”菜单中，选择“实例设置”→“获取系统日志”。

此时会打开一个窗口，然后显示从虚拟服务器得到的系统日志，这些日志通常在启动期间显示在一台物理监视器上（见图3-15）。

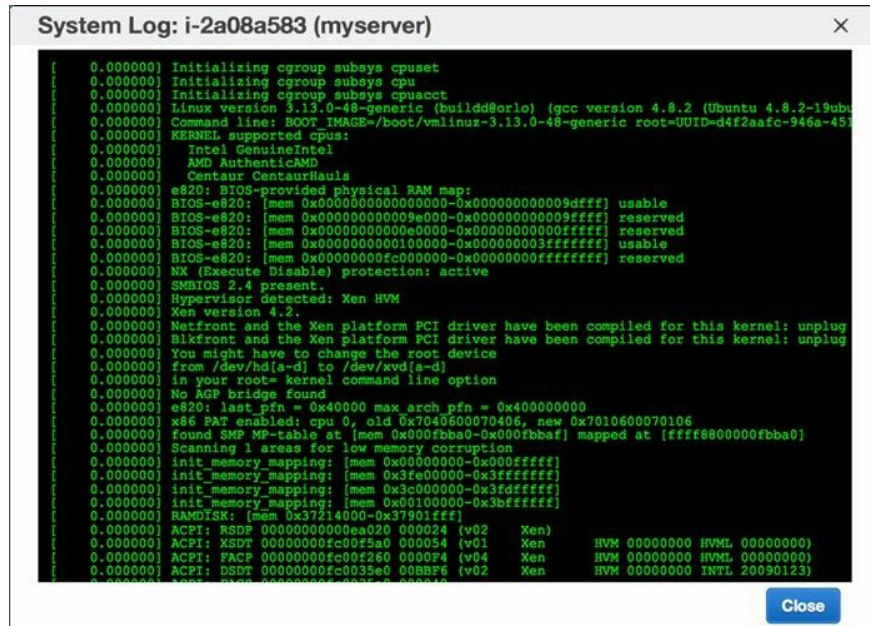


图3-15 在日志的帮助下调试一台虚拟服务器

这是一个简单、有效的访问用户的服务器系统日志，并且它不需要SSH连接。注意，在日志查看器上显示一条日志信息可能会需要花费几分钟。

## 3.2.2 监控虚拟服务器的负载

AWS能帮助用户回答这样的问题：“我的虚拟服务器利用情况是否接近了它的最大容量？”按下面步骤来打开服务器的指标。

- (1) 在主导航栏中打开EC2，然后从子菜单中选择“实例”。
- (2) 点击表中的一行以选择正在运行的虚拟服务器。
- (3) 选择右下角的“监控”标签页。
- (4) 点击“网络输入”来查看详细信息。

用户将看到一张图，它展示了虚拟服务器的流入网络流量的利用率，如图3-16所示。有一些关于CPU使用量、网络使用量和硬盘使用量的指标，但是没有内存使用量的指标。如果用户使用基本监控，这些指

标每5 min更新一次；如果用户对自己的虚拟服务器启用详细监控，这些指标每1 min更新一次。对一些实例类型来说，详细监控会产生费用。

指标和日志将帮助用户监控和调试自己的虚拟服务器。这两个工具都能帮助用户确保自己在以高效率的方式提供高质量的服务。

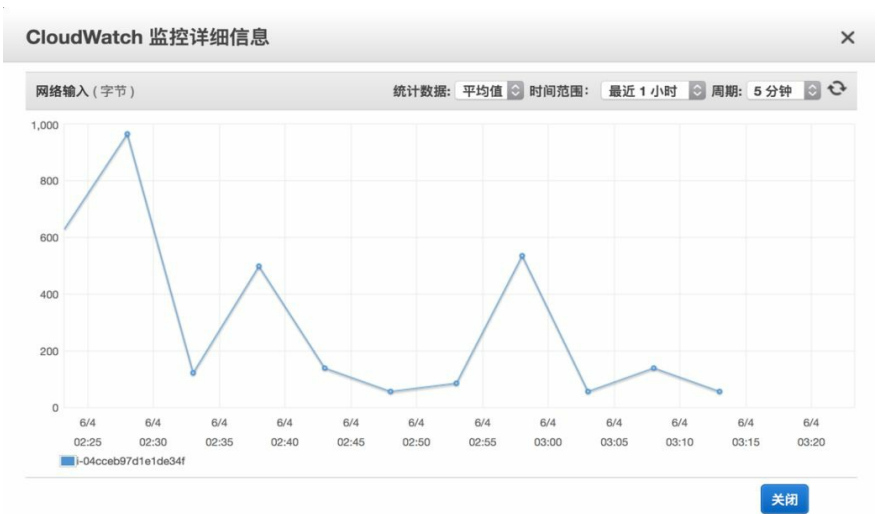


图3-16 使用CloudWatch指标深入分析一台虚拟服务器的流入网络流量

### 3.3 关闭虚拟服务器

为避免产生费用，用户应该总是关闭不用的虚拟服务器。用户可以使用以下4个操作来控制一台虚拟服务器的状态。

- 开启 —— 用户总是能够开启一台停止的虚拟服务器。如果用户需要创建一台全新的服务器，就需要启动一台新的虚拟服务器。
- 停止 —— 用户总是能够停止一台正在运行的虚拟服务器。一台停止了的虚拟服务器不会被收取费用，并且可以再次被开启。如果用户在使用网络附加存储，用户的数据将被保存。一个停止了的虚拟服务器不会产生费用，除了网络附加存储这样的附加资源外。
- 重启 —— 如果用户需要重启自己的虚拟服务器，这个操作非常有帮助。用户不会在重启时丢失任何数据，而且所有的软件在重启后仍会保持被安装了的状态。
- 终止 —— 终止一台虚拟服务器意味着删除它。用户不能再次开启一台已经终止了的虚拟服务器。虚拟服务器被删除了，同时被删除的还有其依赖项（如网络附加存储）和公有与私有IP地址。被终止了的虚拟服务器不会再产生费用。

#### 警告

停止与终止 一台虚拟服务器的区别很重要。用户可以启动一台已经停止的虚拟服务器，但却不能启动一台已经终止的虚拟服务器。如果用户终止了一台虚拟服务器，则意味着把它删除了。

图3-17所示为采用流程图展示了停止与终止一台虚拟服务器的区别。

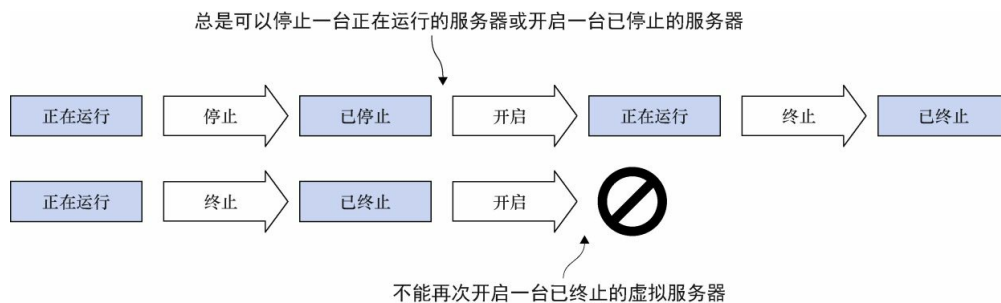




图3-17 停止与终止一台虚拟服务器的区别

停止或终止不用的虚拟服务器能省钱且防止从AWS收到意外的账单。如果用户为一个短期任务启动一台虚拟服务器，别忘了创建一个终止提醒。当用户终止了一台虚拟服务器之后，这台服务器就不再可用，而且最终会从虚拟服务器列表中消失。

#### 资源清理

终止在本章开始时启动的虚拟服务器myserver。

- （1）在主导航栏中打开EC2，在子菜单中选择“实例”。
- （2）点击表中的一行以选择正在运行的这台虚拟服务器（myserver）。
- （3）在“操作”菜单中，选择“实例状态”→“终止”。

## 3.4 更改虚拟服务器的容量

用户总是可以更改一台虚拟服务器的容量。这是云的优势之一，它给了用户垂直扩展的能力。如果用户需要更多的计算能力，还可以增加服务器的容量。

在本章中，我们将学习如何更改一台正在运行的虚拟服务器的容量。首先，让我们按下面的步骤来启动一台小的虚拟服务器。

- (1) 打开AWS管理控制台，选择EC2。
- (2) 打开向导，点击“启动实例”按钮来启动一台新的虚拟服务器。
- (3) 选择“Ubuntu Server 16.04 LTS(HVM)”作为虚拟服务器的AMI。
- (4) 选择实例类型为t2.micro。
- (5) 点击“审核和启动”来启动虚拟服务器。
- (6) 检查新虚拟服务器的汇总信息，然后点击“启动”按钮。
- (7) 选择“选择现有密钥对”选项，选择密钥对mykey，然后点击“启动实例”。
- (8) 切换到EC2实例概览，然后等待新虚拟服务器的状态变为Running。

我们已经启动了一台实例类型为t2.micro的虚拟服务器。这是AWS上可用的最小的虚拟服务器之一。

使用SSH连接到我们的服务器上，如3.3节所述，然后执行`cat /proc/cpuinfo`以及`free -m`来获取服务器的CPU和内存信息。输出应该如下所示：

```
$ cat /proc/cpuinfo
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 62
model name    : Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz
stepping      : 4
microcode     : 0x416
cpu MHz       : 2500.040
cache size    : 25600 KB
[...]

$ free -m
              total        used        free      shared    buffers     cached
Mem:           992         247         744           0           8         191
-/+ buffers/cache:          48         944
Swap:           0           0           0
```

我们的虚拟服务器可以使用一个CPU核并提供992 MB内存。

如果用户需要更多的CPU、更多的内存或者更多的网络容量，有很多可以选择的容量。用户甚至可以修改虚拟服务器的实例家族与版本。要增加自己的虚拟服务器的容量，首先要停止它。

- (1) 打开AWS管理控制台，然后选择EC2。
- (2) 在子菜单中选择“实例”，页面会跳转到虚拟服务器的概览。
- (3) 在列表中点击选择正在运行的虚拟服务器。
- (4) 在“操作”菜单中选择“实例状态”→“停止”。

等到虚拟服务器停止后，我们可以更改实例类型：

(1) 在“操作”菜单的“实例设置”中选择“更改实例类型”。如图3-18所示，将打开一个对话框，可以在这个对话框中为虚拟服务器选择新的实例类型。



图3-18 为实例类型选择m3.large来增加虚拟服务器容量

(2) 在“实例类型”中选择m3.large。

(3) 点击“应用”按钮以保存所做的更改。

现在我们已经更改了虚拟服务器的容量，并准备好再次开启它了。

#### 警告

启动一台实例类型为m3.large的虚拟服务器将会产生费用。如果想知道一台m3.large的虚拟服务器的当前每小时价格，可以访问AWS官方网站。

要做到这一点，选择虚拟服务器并且在“操作”菜单中的“实例状态”下选择“启动”。虚拟服务器将会有更多的CPU、更多的内存和更多的网络能力。公有与私有IP地址也发生了变化。获取新的公有IP地址用以通过SSH重新连接，在虚拟服务器的详细信息视图中可以找到它。

使用SSH连接我们的服务器，然后再次执行`cat /proc/cpuinfo`与`free -m`来获取它的CPU与内存信息。输出结果如下所示：

```
$ cat /proc/cpuinfo
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 62
model name    : Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz
stepping      : 4
microcode     : 0x415
cpu MHz       : 2494.066
cache size    : 25600 KB
[...]
processor      : 1
```

```
vendor_id   : GenuineIntel
cpu family  : 6
model       : 62
model name   : Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz
stepping    : 4
microcode   : 0x415
cpu MHz      : 2494.066
cache size  : 25600 KB
[...]

$ free -m
```

	total	used	free	shared	buffers	cached
Mem:	7479	143	7336	0	6	49
-/+ buffers/cache:		87	7392			
Swap:	0	0	0			

我们的虚拟服务器能够使用两个CPU并提供7479 MB的内存。与之前单个CPU和992 MB的内存容量相比，我们增加了这台服务器的容量。

#### 资源清理

终止实例类型为m3.large的这台虚拟服务器，以停止为它付费。

- (1) 在主导航栏中打开EC2，并在子菜单中选择“实例”。
- (2) 在表中点击这台正在运行的虚拟服务器以选中它。
- (3) 在“操作”菜单中选择“实例状态” → “终止”。

### 3.5 在另一个数据中心开启虚拟服务器

AWS为全球提供数据中心。要使互联网上的请求获得低延迟，为主要用户选择一个最近的数据中心是很重要的。更改数据中心很简单。管理控制台会显示用户目前正在工作的数据中心，位于主导航栏的右边。目前为止，我们使用的都是弗吉尼亚北部的数据中心，称为us-east-1。要变更数据中心，点击弗吉尼亚北部，然后在菜单中选择亚太区域（悉尼）（Sydney）。图3-19展示了如何跳转至Sydney的被称作为ap-southeast-2的数据中心。

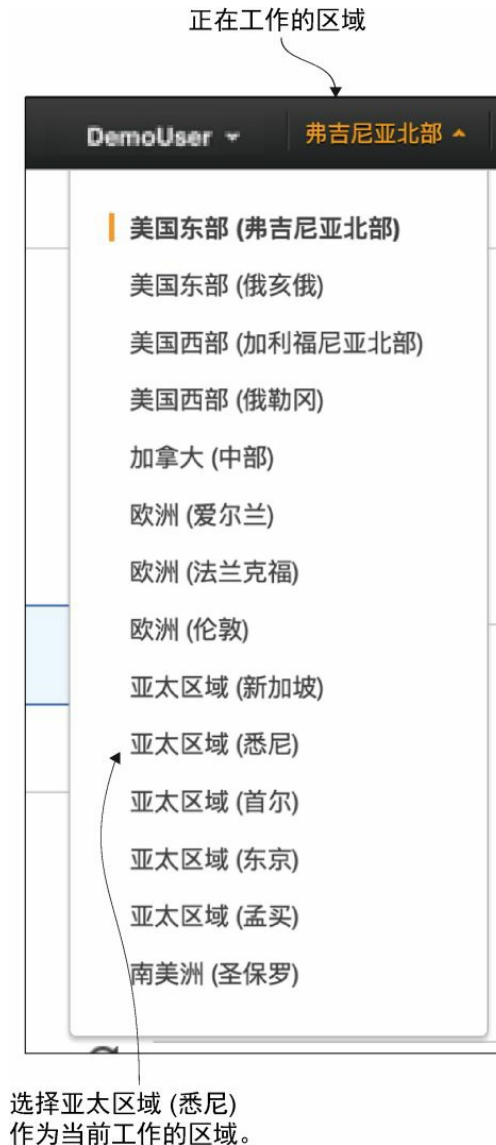


图3-19 在管理控制台中将数据中心从弗吉尼亚北部改为悉尼

AWS将它的数据中心按以下的区域进行分组：

- 亚太区域（东京）（Asia Pacific, Tokyo, ap-northeast-1）；
- 欧洲（法兰克福）（EU, Frankfurt, eu-central-1）；
- 美国东部（弗吉尼亚北部）（US East, N. Virginia, us-east-1）；
- 加拿大（中部）（Canada, Central, ca-central-1）；
- 亚太区域（首尔）（Asia Pacific, Seoul, ap-northeast-2）；
- 欧洲（爱尔兰）（EU, Ireland, eu-west-1）；
- 美国东部（俄亥俄）（US East, Ohio, us-east-2）；
- 南美洲（圣保罗）（South America, Sao Paulo, sa-east-1）；
- 亚太区域（新加坡）（Asia Pacific, Singapore, ap-southeast-1）；
- 欧洲（伦敦）（EU, London, eu-west-2）；
- 美国东部（加利福尼亚北部）（US West, N. California, us-west-1）；
- 亚太区域（悉尼）（Asia Pacific, Sydney, ap-southeast-2）；
- 亚太区域（孟买）（Asia Pacific, Mumbai, ap-south-1）；
- 美国西部（俄勒冈）（US West, Oregon, us-west-2）。

用户可以为大多数AWS服务指定区域。各个区域间是完全独立的，数据不在区域间传输。典型情况下，一个区域由两个或更多位于同一地区的数据中心组成。这些数据中心间有着很好的连接，它们能提供高可用的基础架构，本书稍后会介绍。一些AWS服务，如内容分发网络（content delivery network, CDN）服务以及域名系统（Domain Name System, DNS）服务，是在这些区域外的数据中心之上全球运行的。

当切换到管理控制台的EC2服务后，用户可能想知道为什么EC2概览中没有列出任何密钥对。我们为SSH登录在区域“美国东部（弗吉尼亚北部）”创建了一对密钥。但是，区域间是独立的，所以我们必须为区域亚太区域（悉尼）创建一对新的密钥对。请按以下步骤操作（更多细节参见1.2节）。

- （1）在主导航栏中打开EC2服务，然后在子菜单中选择“密钥对”。
- （2）点击“创建密钥对”，在密钥对名称处输入sydney。
- （3）下载并保存密钥对。

(4) 仅对Windows: 打开PuTTYgen, 然后在“Type of Key to Generate”中选择SSH-2 RSA。点击Load。选择sydney.pem文件并点击打开。在对话框中选择确定。点击Save Private Key。

(5) 仅对Linux和OS X: 在控制台运行`chmod 400 sydney.pem`来改变文件sydney.pem的访问权限。

我们已经准备好在Sydney的数据中心开启一台虚拟服务器了。用户可按以下步骤操作。

(1) 从主导航栏中打开EC2服务, 然后从子菜单中选择“实例”。

(2) 点击“启动实例”, 打开一个向导, 它会开启一台新的虚拟服务器。

(3) 选择Amazon Linux AMI (HVM) machine映像。

(4) 选择t2.micro作为实例类型, 然后点击“审核和启动”的快捷方式来开启一台虚拟服务器。

(5) 点击“编辑安全组”来配置防火墙。更改“安全组名字”为webserver, “描述”为HTTP和SSH。添加一条类型为SSH的规则以及另一条类型为HTTP的规则。对这两条规则, 定义0.0.0.0/0作为源, 从而允许从任何地方访问SSH和HTTP。防火墙配置应该如图3-20所示。点击“审核和启动”按钮。

(6) 点击“启动”, 然后选择sydney作为已经存在的密钥对, 来开启虚拟服务器。

(7) 点击“查看实例”切换到虚拟服务器概览, 然后等待新虚拟服务器启动。

完成了! 一台虚拟服务器在Sydney的数据中心运行起来了。让我们继续在上面安装一个网络服务器。要这样做, 我们必须通过SSH连接到虚拟服务器。从详细信息页抓取虚拟服务器的当前公有IP地址。

打开终端, 输入`ssh -i $PathToKey/sydney.pem ec2-user@$PublicIp`, 使用下载的密钥文件sydney.pem的路径替换\$PathToKey, 使用虚拟服务器详细信息中的公有IP地址替



换\$PublicIp。对关于新主机的认证安全警告回答Yes。

在建立了SSH会话之后，用户可以通过执行`su do yum install httpd -y`安装一个默认网路服务器。要启动这个网络服务器，输入`sudo service httpd start`，然后按Enter键执行这个命令。如果用户打开`http://$PublicIp`并使用自己的虚拟服务器的公有IP替换\$PublicIp，网络浏览器应该会显示一个占位网站。

## Windows

找到在下载新密钥对后创建的sydney.ppk文件，然后双击打开它。PuTTY Pageant应该会在任务栏中显示为一个图标。接下来，启动PuTTY，然后连接到虚拟服务器详细信息中的公有IP地址。对关于新主机的认证安全警告回答Yes，然后输入ec2-user 作为登录名，按Enter键。



图3-20 为Sydney的网络服务器配置防火墙

## 注意

在本章中，我们使用两种不同的操作系统。在本章开始的时候我们启动了一台基于Ubuntu的虚拟服务器，现在使用Amazon Linux，一个基于Red Hat Enterprise Linux的发行版本。这就是要执行不同的命令来安装软件的原因。Ubuntu使用`apt-get`，而Amazon Linux使

用yum。

接下来，我们将关联一个固定公有IP地址到虚拟服务器。

### 3.6 分配一个公有IP地址

在阅读本书时我们已经启动了一些虚拟服务器。每个虚拟服务器都自动连接到一个公有IP地址。但是，每次启动或停止一台虚拟服务器，公有IP地址就改变了。如果想要用一个固定IP地址运行一个应用程序，这样做就不可行了。AWS提供一项服务叫作弹性IP地址（Elastic IP address）来分配固定的公有IP地址。

用户可以使用以下步骤来分配并关联一个公有IP地址到一台虚拟网络服务器上。

- （1）打开管理控制台，并打开EC2。
- （2）从子菜单中选择“弹性IP”。用户将看见一个公有IP地址的总览，如图3-21所示。



图3-21 用户的账号在当前区域关联的公有IP地址总览

- （3）点击“分配新地址”来分配公有IP地址。
- 现在我们可以为自己选择的一台虚拟服务器关联公有IP地址。

(1) 选择自己的公有IP地址，然后在“操作”菜单中选择“关联地址”，将显示一个图3-22所示的对话框。

(2) 在“实例”项中输入虚拟服务器的实例ID。网络服务器是此时唯一正在运行的虚拟服务器，所以可以输入*i-* 并使用自动完成来选择服务器ID。

(3) 点击“关联”来完成这一流程。

现在我们的虚拟服务器可以通过在本节开头分配的公有IP地址来访问了。将浏览器指向这一IP地址，我们将看到如3.5节所做的占位页面。

如果用户必须确保自己的应用的端点不变化，甚至当不得不替换后台的虚拟服务器时，分配公有IP地址变得很有用。例如，假设虚拟服务器A正在运行并且关联了一个弹性IP地址。接下来的步骤能让用户将虚拟服务器替换成一台新的而不需要中断服务。

(1) 启动一台新的虚拟服务器B用以替换正在运行的服务器A。

(2) 在虚拟服务器B上安装并启动应用以及所有的依赖项。

(3) 从虚拟服务器A解除弹性IP关联，并将它关联到虚拟服务器B。

选择虚拟服务器的ID。



地址 > 关联地址

关联地址

选择要与此弹性 IP 地址 (52.64.75.47) 关联的实例或网络接口

资源类型 ☒ 实例 ☐ 网络接口

实例

私有 IP

重新关联 ☐ 允许重新关联已附加的弹性 IP

**警告**  
如果您将弹性 IP 地址与您的实例关联，则将释放当前弹性 IP 地址。了解有关弹性 IP 地址的更多信息。

\* 必填

取消 关联

图3-22 为网络服务器关联一个公有IP地址

使用弹性IP地址的请求将被路由到虚拟服务器B，而服务不会发生中断。

用户也可以使用多个网络接口来关联多个公有IP地址到一台虚拟服务器，正如3.7节中所描述的。如果用户需要在同一端口运行不同的应用或者不同的网站使用一个唯一的固定的公有IP地址，这会很有用。

**警告**

IPv4地址是稀缺资源。为了防止浪费弹性IP地址，AWS将对没有关联到任何服务器的弹性IP地址收费。我们将在3.7节结束时清除所分配的IP地址。

### 3.7 向虚拟服务器添加额外的网络接口

除了管理公有IP地址，用户还能够控制自己的虚拟服务器的网络接口。用户可以向一台虚拟服务器添加多个网络接口，并且控制关联到这些网络接口的私有IP地址和公有IP地址。用户可以使用额外的网络接口关联第二个公有IP地址到自己的虚拟服务器上。

用户可以按下面的步骤来为自己的虚拟服务器创建一个额外的网络接口（见图3-23）。

- （1）打开管理控制台并跳转至EC2服务。
- （2）在子菜单中选择“网络接口”。
- （3）点击“创建网络接口”，会弹出一个对话框。
- （4）输入2nd interface 作为描述。

（5）选择自己的虚拟服务器的子网作为新的网络接口的子网，用户能在实例总览中自己的服务器的详细视图找到这个子网信息。

- （6）让“私有IP”地址保持为空。



图3-23 为虚拟服务器创建额外的网络接口

- （7）选择在其描述中有webserver的“安全组”。
- （8）点击“是，请创建”。

当新的网络接口的状态变为Available，用户可以将它附加到自己的虚拟服务器。选择新的网络接口**2nd Interface**，然后在菜单中选择“附加”，会弹出一个对话框，如图3-24所示。选择正在运行的虚拟服务器的ID，然后点击“附加”按钮。



图3-24 附加额外的网络接口至虚拟服务器

我们已经附加了一个额外的网络接口至自己的虚拟服务器。接下来，我们将关联一个额外的公有IP地址到这个额外的网络接口。要这样做，先记录在总览中显示的这个额外网络接口的网络接口ID，然后按照下面的步骤进行操作。

- (1) 打开管理控制台且转到EC2服务。
- (2) 从子菜单中选择“弹性IP”。
- (3) 点击“分配新地址”来分配一个新的公有IP地址，如在3.6节中所做的。
- (4) 从“操作”菜单中选择“关联地址”，然后将它链接到刚才在“网络接口”中输入网络接口ID所创建的额外的网络接口（见图3-25）。

现在虚拟服务器可以通过两个不同的公有IP地址来访问了。这样用户可以根据公有IP地址提供两个不同的网站服务。我们需要配置网络服务器来根据公有IP地址来应答请求。

在使用SSH连接到我们的虚拟服务器，并且在终端输入**ifconfig**之后，就能看见自己的新网络接口附加到了虚拟服务器上，代码如下所示：

```
$ ifconfig
eth0      Link encap:Ethernet HWaddr 12:C7:53:81:90:86
          inet addr:172.31.1.208 Bcast:172.30.0.255 Mask:255.255.255.0
          inet6 addr: fe80::10c7:53ff:fe81:9086/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:62185 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9179 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:89644521 (85.4 MiB) TX bytes:582899 (569.2 KiB)

eth1      Link encap:Ethernet HWaddr 12:77:12:53:39:7B
          inet addr:172.31.4.197 Bcast:172.30.0.255 Mask:255.255.255.0
          inet6 addr: fe80::1077:12ff:fe53:397b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1256 (1.2 KiB) TX bytes:1374 (1.3 KiB)

[...]
```

选择你刚才创建的网络接口

地址 > 关联地址

### 关联地址

选择要与此弹性 IP 地址 (52.64.76.47) 关联的实例或网络接口

资源类型 ☐ 实例 ☒ 网络接口

网络接口

私有 IP

重新关联 ☐ 允许重新关联已附加的弹性 IP

**警告**  
如果您将弹性 IP 地址与您的实例关联, 则将释放当前弹性 IP 地址。了解有关弹性 IP 地址的更多信息。

\* 必填 取消 关联

图3-25 将公有IP地址关联到额外的网络接口上

每个网络接口都连接到一个私有IP地址和一个公有IP地址。我们需要配置网络服务器来根据IP地址提供不同的网站。虚拟服务器不知道任何关于它的公有IP地址的事, 但我们可以根据私有IP地址来区分请求。

首先需要两个网站。在悉尼的虚拟服务器上通过SSH运行以下命令来下载两个简单的占位网站:

```
$ sudo -s
```



```
$ mkdir /var/www/html/a
$ wget -P /var/www/html/a https://raw.githubusercontent.com/AWSInAction/\
code/master/chapter3/a/index.html
$ mkdir /var/www/html/b
$ wget -P /var/www/html/b https://raw.githubusercontent.com/AWSInAction/\
code/master/chapter3/b/index.html
```

接下来需要配置网络服务器来根据IP地址分发网站。  
在/etc/httpd/conf.d下添加一个名为a.conf的文件，将IP地址从172.31.x.x修改为ifconfig 输出的网络接口eth0的IP地址：

```
<VirtualHost 172.31.x.x:80>
  DocumentRoot /var/www/html/a
</VirtualHost>
```

重复同样的操作过程，在/etc/httpd/conf.d下创建一个名为b.conf的内容如下的配置文件。将IP地址从172.31.y.y 修改为ifconfig 输出的网络接口eth1的IP地址：

```
<VirtualHost 172.31.y.y:80>
  DocumentRoot /var/www/html/b
</VirtualHost>
```

要激活新的网络服务器配置，通过SSH执行**sudo service httpd restart**。在管理控制台切换至弹性IP总览。复制两个公有IP地址，然后在网络浏览器中分别打开它们。根据访问的不同公有IP地址，用户应该得到回应“Hello A!”或“Hello B!”。这样用户能够根据用户访问的不同公有IP地址来提供两个不同的网站。

#### 资源清理

是时候做一些清理工作了。

- (1) 终止虚拟服务器。
- (2) 转到“网络接口”，然后选择并且删除网络接口。

（3）切换至弹性IP，然后选择并从“操作”菜单中点击“释放地址”，释放两个公有IP地址。

就是这样，一切都清理好了，准备进入下一节。

### 3.8 优化虚拟服务器的开销

通常用户按需在云中启动自己的虚拟服务器来获取最大的灵活性。用户可以随时启动或停止一个按需实例，而且会按实例（虚拟服务器）运行的小时数来结算费用。如果想省钱，有两个选项：竞价型实例或预留实例。这两个选项都能够帮助用户减少开销，但是这样会降低灵活性。对于竞价型实例，可以对AWS数据中心中未使用的容量出价，价格基于供给与需求。如果需要使用一台虚拟服务器超过一年，可以使用预留实例，同意支付给定时间段的费用并提前获取折扣。表3-2展示了这些选项之间的不同点。

表3-2 按需、预留及竞价型虚拟服务器的不同点

	按需	预留	竞价型
价格	高	中	低
灵活性	高	低	中
可靠性	中	高	低

#### 3.8.1 预留虚拟服务器

预留一台虚拟服务器意味着承诺使用在指定数据中心的一台指定类型的虚拟服务器。无论这台预留虚拟服务器在运行还是不在运行，用户都必须为它支付费用。作为回报，用户得到最多可达到60%的价格优惠。在AWS上，如果想预留一台虚拟服务器，可以选择以下选项中的一个：

- 无前期费用，1年使用期；
- 部分前期费用，1年或3年使用期；

- 全部前期费用，1年或3年使用期。

表3-3展示了对配有1台CPU、3.75 GB内存和4 GB SSD的虚拟服务器（称为m3.medium），这意味着什么呢？

表3-3 虚拟服务器（m3.medium）的潜在可节约成本

	月成本（美元）	前期成本（美元）	实际月成本（美元）	与按需相比节省了
按需	48.91	0.00	48.91	
无前期费用，1年使用期	35.04	0.00	35.04	28%
部分前期费用，1年使用期	12.41	211.00	29.99	39%
全部前期费用，1年使用期	0.00	353.00	29.42	40%
部分前期费用，3年使用期	10.95	337.00	20.31	58%
全部前期费用，3年使用期	0.00	687.00	19.08	61%

在AWS上用户可以使用预留虚拟服务器来以灵活性换取成本减少。但是还有更多选择。如果用户有一台虚拟服务器的预留（一个预留实例），这台虚拟服务器的容量在公有云中是为用户预留的。为什么这很重要？假设在一个数据中心里，对虚拟服务器的需求增加了，有可能是因为另一个数据中心坏了，而许多AWS客户不得不启动新的虚拟服务器来替换他们坏了的那些服务器。在这种罕见的情况下，按需虚拟服务器的订单堆积起来，有可能变成很难启动一台新的虚拟服务器。如果用户计划构建一个高可用的跨多个数据中心的设置，应该考虑预留最小

的能保持自己的应用运行的容量。我们推荐开始时使用按需服务器，然后切换到按需与预留服务器混合的模式。

### 3.8.2 对未使用的虚拟服务器竞价

除了预留虚拟服务器之外，还有另外一个选项可以降低成本——竞价型实例。用户使用一个竞价型实例，对AWS云中未使用的容量进行竞价。一个现货交易市场是指一个标准产品在交易后马上交货的市场。在这个市场上的产品价格依赖于供给与需求。在AWS竞价市场，交易的产品是虚拟服务器，它们是通过启动一台虚拟服务器来提供的。

图3-26展示了指定实例类型的一台虚拟服务器的价格。如果在指定数据中心的一台指定虚拟服务器的当前竞价型实例价格比自己的最高价格低，用户的竞价型实例请求将被满足，一台虚拟服务器将启动。如果当前的竞价型实例价格超出用户的竞价，用户的虚拟服务器将在2 min后被AWS终止（不是停止）。

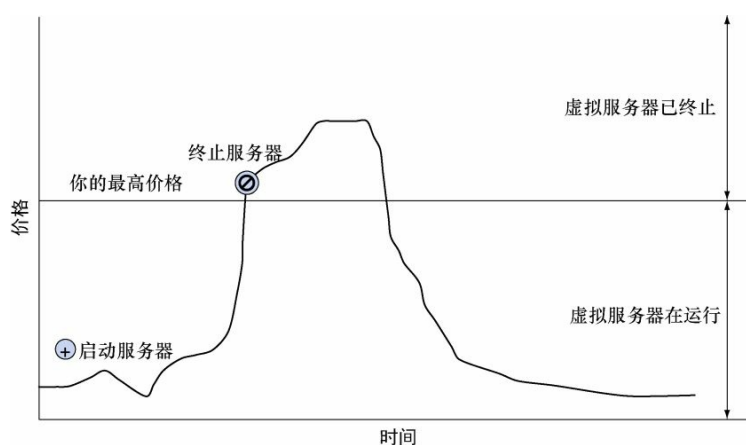


图3-26 虚拟服务器现货交易市场的功能

竞价价格可能更高或更低的灵活性取决于虚拟服务器的大小以及所在的数据中心。我们看见过竞价型实例价格只有按需价格的10%，也看见过竞价型实例价格甚至超过按需价格。一旦竞价型实例价格超过用户的竞价，用户的服务器将在2 min内被终止。用户不应该将竞价实例用在类似网络或邮件服务器上，但是可以用它们来运行异步任务，如数据分析或者对媒体资产进行编码。用户甚至可以用竞价实例来检查自己的网站的破损连接，如在3.1节所做的，因为这不是一个时间要求严格的

任务。

让我们来启动一台使用竞价型实例市场优惠价格的新的虚拟服务器。首先用户必须将订单提交到现货交易市场。图3-27展示了请求虚拟服务器的开始点。我们可以在主导航栏中选择EC2服务，然后从子菜单中选择“竞价请求”进入。点击“定价历史记录”，可以看到虚拟服务器的价格；不同的服务器大小及不同数据中心的历史价格都能看到。



图3-27 请求一个竞价型实例

在3.1节中，我们启动了一台虚拟服务器。请求一个竞价型实例的步骤大致相同。点击“请求竞价型实例”按钮打开向导。选择“Ubuntu Server 16.04 LTS (HVM)”作为虚拟服务器的操作系统。

图3-28所示的步骤可以用来选择虚拟服务器的尺寸。用户不能启动实例类型为t2家族的竞价型实例，因此像t2.micro这样的实例类型是不可用的。

#### 警告

通过请求竞价型实例启动一台实例类型m3.medium的虚拟服务器将会产生费用。在下面的例子中最大价格（竞价）为0.07美元/小时。

选择最小可用的虚拟服务器类型m3.medium，然后点击“选择”。

下一步，如图3-29所示，配置虚拟服务器的详细信息以及竞价型实例请求。设置下面参数。

- (1) 设置竞价型实例请求的目标容量为1。
- (2) 选择0.070作为虚拟服务器的最高价。这是该服务器尺寸的按需价格。
- (3) 选择默认网络，使用IP地址范围为172.30.0.0/16。
- (4) 看一下当前竞价价格项，然后搜索最低的价格，选择相应的子网。

点击“审阅”完成向导。我们将看见我们所做的所有设置的总结。点击“启动”完成请求竞价型实例。

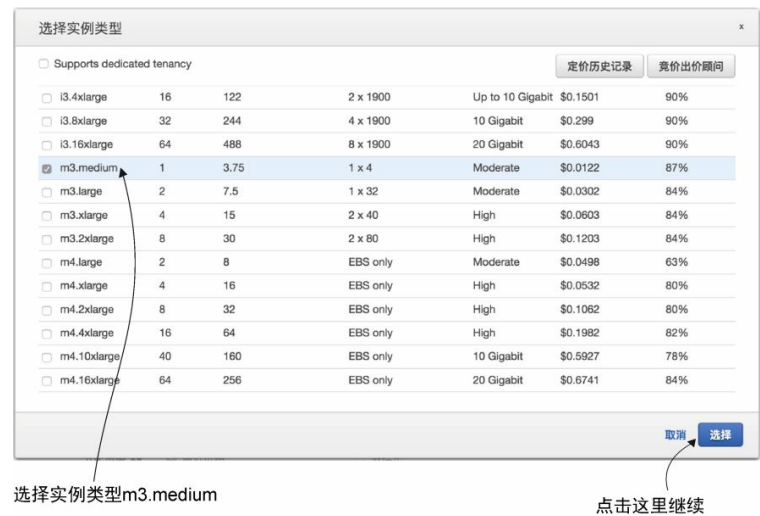


图3-28 选择竞价型服务器尺寸

这次申请竞价的数量

指定的镜像 (AMI) 和实例类型

目标容量 1

AMI Amazon Linux AMI 2017.03.0.20170417 x86\_64 HVM

实例类型 c3.large (2 vCPU, 3.75 GiB, 2 x 16)

分配策略 最低价格 (自动选择最便宜的可用区和实例类型)

网络 vpc-2cc23649 (172.31.0.0/16) (默认)

可用区 ap-southeast-2a

子网 subnet-9723i

最高价 设定您的最高价 (每实例 / 小时) \$ 0.07

没有找到您在查找的吗? 使用旧的竞价型实例启动向导。

取消 下一步

网络与访问控制保持默认值

你的服务器的最高价格

点击这里继续

图3-29 选择虚拟服务器详细信息，然后指定一个最大小时价格

完成了向导的所有步骤，虚拟服务器竞价请求就被放到了竞价型实例市场上。点击“竞价请求”子菜单，将把用户带回竞价型实例请求总览。用户应该看见图3-30所示的一个竞价型实例请求。可能需要几分钟时间请求才会被满足。看一下虚拟服务器请求的状态：因为竞价型实例市场不可预测，请求失败也是有可能的。如果发生了请求失败，重复上面的操作过程来生成另一个请求，并且选择另一个子网来启动虚拟服务器。

请求竞价型实例 Spot Advisor 操作 定价历史记录

请求类型: all 状态: all 按关键字搜索

查看 2 个请求中的 1 到 2

请求 ID	请求类型	实例类型	状态	容量	状态	持久性	创建于
sir-hri8jwvj	instance	c3.large	open	-	pending-eval...	one-time	a few seconds a
sfr-70b572cc-3d52...	fleet	c3.large	active	0 of 1	pending_fulfill...	request	a few seconds a

等待状态变为fulfilled

图3-30 等待竞价型实例请求被满足且虚拟服务器被启动

如果请求的状态变为fulfilled，就说明一台虚拟服务器被启动了。用户可以通过子菜单切换到“实例”看一下，会在虚拟服务器总览的实例列表中找到一台正在运行或启动的实例。我们成功地启动了一台竞价型



## 实例的虚拟服务器！

### 资源清理

终止实例类型为m3.medium的虚拟服务器，以停止为它付费。

- （1）从主导航栏中打开EC2服务，然后在子菜单中选择“实例”。
- （2）在表格中点击行，选中正在运行的虚拟服务器。
- （3）在“操作”菜单中，选择“实例状态”→“终止”。
- （4）切换到“竞价请求”总览，再次确认竞价型实例请求已经取消了。如果没有取消，选择这个竞价型实例请求，然后点击“取消竞价请求”。

## 3.9 小结

- 可以在启动一台虚拟服务器时选择操作系统。
- 使用日志与指标有助于用户监控和调试一台虚拟服务器。
- 改变虚拟服务器的尺寸可以灵活改变CPU、内存及存储的数量。
- 可以从遍布全球的不同区域（由多个数据中心组成）启动虚拟服务器。
- 分配及关联一个公有IP地址到虚拟服务器能够灵活地替换一台虚拟服务器，而不需要改变公有IP地址。
- 可以通过预留虚拟服务器或在虚拟服务器竞价型实例市场上对未使用的容量进行竞价来节约成本。

## 第4章 编写基础架构：命令行、SDK和CloudFormation

### 本章主要内容

- 理解“基础设施即代码”的思想
- 使用CLI来启动虚拟服务器
- 使用Node.js上的JavaScript SDK来启动虚拟服务器
- 使用CloudFormation来启动虚拟服务器

想象一下，你想要把房间照明作为一项服务来提供。要使用软件关闭房间灯光，需要一个硬件设备，如一个可以切断电流的继电器。这个硬件设备必须有某种接口让你能通过软件向它发送开灯和关灯这样的指令。使用一个继电器及其接口，就可以将房间照明变成一种服务。这一概念同样适用于虚拟服务器。如果想通过软件启动一台虚拟服务器，需要能处理并满足请求的硬件设备。AWS提供通过接口来控制的基础架构，叫作应用编程接口（application programming interface, API）。用户能通过API控制AWS的每一部分。用户可以使用大多数编程语言、命令行和更复杂的工具的SDK调用这些API。

#### 不是所有示例都包含在免费套餐中

本章中的示例不都包含在免费套餐中。当一个示例产生费用时，会显示一个特殊的警告消息。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

在AWS上，一切操作都可以通过API来控制。用户通过HTTPS协议调用REST API来与AWS交互，如图4-1所示。一切操作都可以通过API提供。例如，用户可以通过一个API调用启动一台服务器，创建1 TB存储，或通过API启动一个Hadoop集群。这里说的“一切”真的包含了云上的所有操作。我们需要一些时间来理解这样的概念。当读完本书时，读者可能会感到遗憾：为什么现实世界不能像云计算那样简单。下面让我

们看看API是怎么工作的。

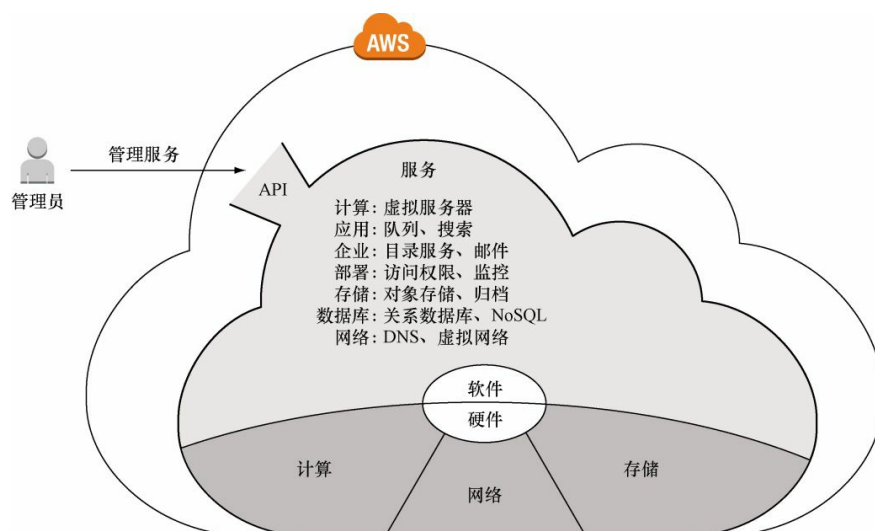


图4-1 调用REST API与AWS交互

要列出S3 对象存储里的所有文件，可以向API端点发送一个GET 请求：

```
GET / HTTP/1.1
Host: BucketName.s3.amazonaws.com
Authorization: [...]
```

请求的响应大概如下所示：

```
HTTP/1.1 200 OK
x-amz-id-2: [...]
x-amz-request-id: [...]
Date: Mon, 09 Feb 2015 10:32:16 GMT
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
[...]
```

使用底层的HTTPS请求直接调用API不太方便。另一种简单的方法

是，使用命令行接口或SDK来和AWS交互，正如在本章中所学的那样。API是这些工具的基础。

## 4.1 基础架构即代码

“基础架构即代码”表达了使用高级编程语言来控制IT系统的思想。在软件开发中，自动化测试、代码库和构建服务器提高了软件工程的质量。如果用户的基础架构可以当作代码来对待，用户就能够对自己的基础架构代码和自己的应用程序代码使用相同的技术。最终，用户将可以使用自动化测试、代码库和构建服务器来改善基础架构的质量。

### 警告

不要混淆基础架构即代码与基础架构即服务（IaaS）的概念！IaaS指的是按照使用量进行付费的租用服务器、存储和网络的业务模式。

### 4.1.1 自动化和DevOps运作

DevOps（development operations）是软件开发驱动的一个方法，以便让开发和运维更加紧密地配合。其目标是能快速发布开发好的软件，并且没有损失质量。因而开发与运维的沟通与合作就变得必需了。

只有把代码修改和代码部署的过程完全自动化，才有可能在一天内部署多次代码。如果用户提交源代码到代码库中，源代码将被自动构建并使用自动化测试进行测试。如果构建结果通过了测试，它会自动安装到测试环境。接下来可能触发一些集成测试。集成测试通过后，这个更改会被传送入产品。但是这还不是流程的结束，现在用户还需要仔细监控系统并实时分析日志，以确保更改是成功的。

如果用户的基础架构是自动化的，用户可以为每一个提交到代码库的更改启动一个新系统，用来单独运行与同一时刻提交到代码库中的其他代码隔离的集成测试。任何时候有代码变动，将创建一个新系统（服务器、数据库和网络等）来单独运行这一变动。

### 4.1.2 开发一种基础架构语言：JIML

为了易于详细理解基础架构即代码，人们开发了一种新语言来描述基础架构：JSON基础架构标记语言（JSON Infrastructure Markup Language, JIML）。图4-2描述了将要创建的基础架构。

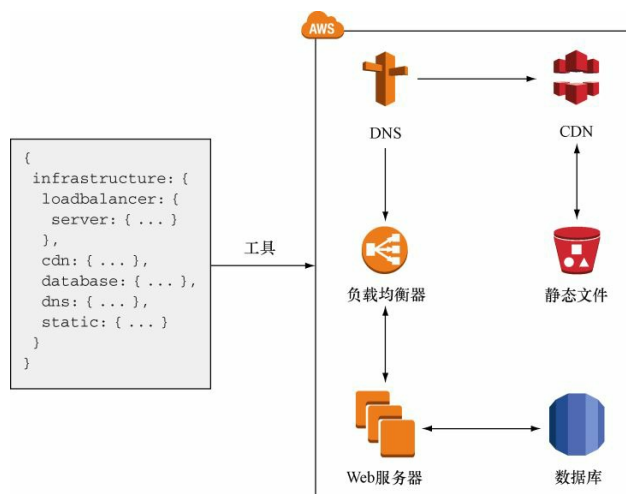


图4-2 从JIML蓝图到基础架构：基础架构自动化

这个基础架构包含以下内容：

- 负载均衡器（LB）；
- 虚拟服务器；
- 数据库（DB）；
- DNS域名入口；
- 内容分发网络（CDN）；
- 静态文件存储桶。

为了减少语法问题，我们让JIML基于JSON格式。代码清单4-1所示的JIML程序创建了图4-2所示的基础架构。\$ 表示指向一个ID的引用。

代码清单4-1 用JIML描述基础架构

```
{
  "region": "us-east-1",
  "resources": [{
    "type": "loadbalancer",
    "id": "LB",
    "config": {
      "server": {
        "cpu": 2,
        "ram": 4,
```

```

        "os": "ubuntu",
        "waitFor": "$DB"
    },
    "servers": 2
}
}, {
    "type": "cdn",
    "id": "CDN",
    "config": {
        "defaultSource": "$LB",
        "sources": [{
            "path": "/static/ *",
            "source": "$BUCKET"
        }]
    }
}, {
    "type": "database",
    "id": "DB",
    "config": {
        "password": "****",
        "engine": "MySQL"
    }
}, {
    "type": "dns",
    "config": {
        "from": "www.mydomain.com",
        "to": "$CDN"
    }
}, {
    "type": "bucket",
    "id": "BUCKET"
}]
}

```

JSON是怎么被转换成AWS API调用的呢？

(1) 解析JSON输入。

(2) JIML解释器将资源和它们的依赖项连接起来，创建一张依赖图。

(3) JIML解释器从底层（叶子）到顶层（根）遍历依赖图中的树，然后产生一个线性的命令流。这些命令由一个伪语言来表达。



(4) 然后JIML运行环境将这些伪语言的命令翻译成AWS API调用。

让我们来看看由JIML解释器创建的依赖图，如图4-3所示。

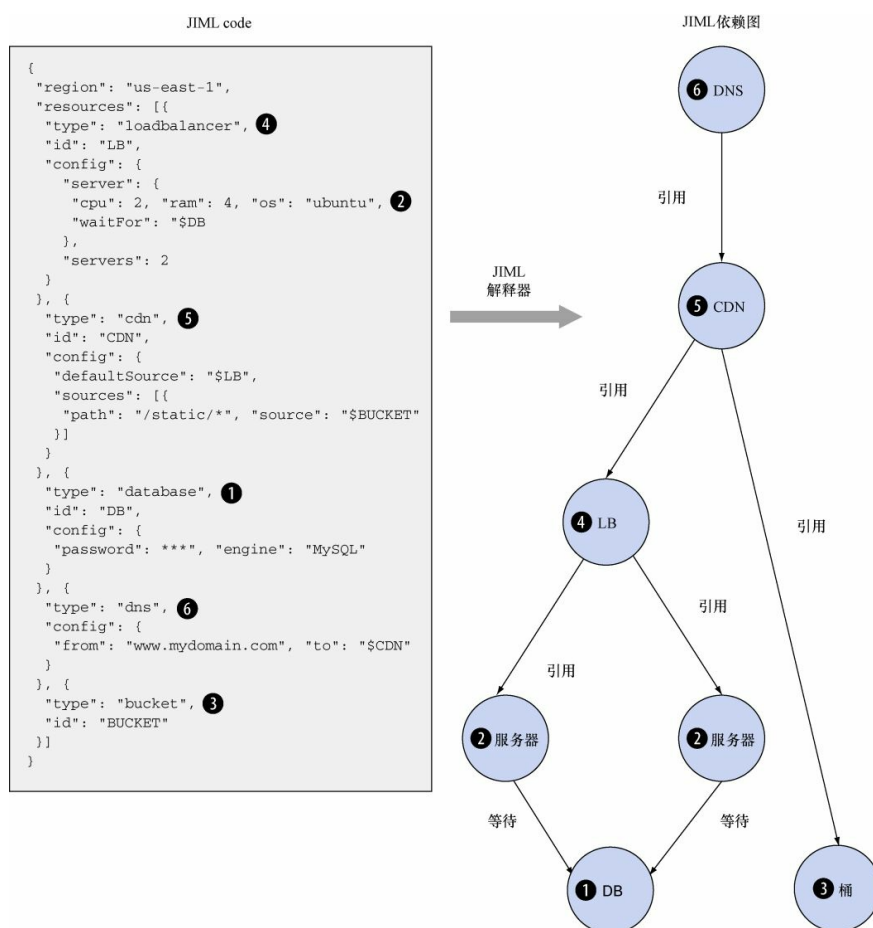


图4-3 JIML解释器确定了资源创建的顺序

自底向上从左到右遍历这张依赖图，底部的节点DB ①和bucket ③没有子节点。没有子节点的节点就没有依赖项。服务器②节点依赖于DB ①节点。LB④依赖于服务器②节点。CDN ⑤节点依赖于LB ④节点和bucket ③节点。最后，DNS ⑥节点依赖于CDN 节点。

JIML解释器把依赖图变成一个线性的使用伪语言的命令流，详见代码清单4-2。这个伪语言代表了用正确的顺序创建所有资源所需要的步骤。底部的节点没有依赖项，所以最容易创建：这就是它们首先被创建的原因。

```
$DB = database create {"password": "****", "engine": "MySQL"}    <--创建数据库
$BUCKET = bucket create {}

await $DB
$SERVER1 = server create {"cpu": 2, "ram": 4, "os": "ubuntu"}    <--创建服务器
$SERVER2 = server create {"cpu": 2, "ram": 4, "os": "ubuntu"}

await [$SERVER1, $SERVER2]    <--等待依赖
$LB = loadbalancer create {"servers": [$_SERVER1, $_SERVER2]}    <--创建负载均衡器

await [$LB, $BUCKET]
$CDN = cdn create {...}    <--创建CDN

await $CDN
$DNS = dns create {...}    <--创建DNS 入口

await $DNS
```

最后的步骤——把伪语言命令翻译成AWS API调用——这里省略了。我们已经学习了基础架构即代码所需的一切：都与依赖相关。

现在我们理解了依赖关系对于基础架构即代码有多重要，让我们来看看如何使用命令行创建基础架构。命令行是实现基础架构即代码的一种工具。

## 4.2 使用命令行接口

AWS命令行接口（command-line interface, CLI）是一个从命令行使用AWS的便捷的方法。它运行在Linux、Mac和Windows上，是用Python写的。它为所有AWS服务提供了一个统一的访问接口。除非另作说明，否则命令的输出都是JSON格式的。

现在我们将安装和配置CLI，之后，就可以开始着手使用了。

### 4.2.1 安装CLI

怎样继续要看用户的操作系统。

#### 1. Linux和Mac OS X

CLI需要Python（2.6.5或更高、2.7.x或更高、3.3.x或更高或3.4.x和更高）以及pip。pip安装Python程序包的推荐工具。要检查Python版本，在终端运行`python -version`。如果用户没有安装Python或者版本太旧，需要找到一个代替方法来安装Python。要查看是否安装了pip，在终端上运行`pip --version`。如果显示了版本，说明已经安装了；否则，执行下面的命令来安装pip：

```
$ curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py"
$ sudo python get-pip.py
```

再在终端上运行`pip --version`来验证pip安装。现在是时候来安装AWS CLI了：

```
$ sudo pip install awscli
```

在终端运行`aws --version` 来验证AWS CLI安装。

## 2. Windows

下面的步骤引导用户在Windows上使用MSI安装程序来安装AWS CLI。

- (1) 下载AWS命令行接口（32位或64位）MSI安装程序。
- (2) 运行下载好的安装程序，然后按照安装向导的提示安装CLI。
- (3) 用管理员身份运行PowerShell，在“开始”菜单中找到PowerShell项，然后在关联菜单中选择以管理员身份运行。
- (4) 在PowerShell中输入`Set-ExecutionPolicy Unrestricted`，然后按Enter键执行这一命令。这样就能执行我们示例中的未签名的PowerShell脚本。
- (5) 关闭PowerShell窗口，不再需要以管理员身份工作了。
- (6) 通过“开始”菜单中的PowerShell项运行。
- (7) 在PowerShell中执行`aws --version`，以验证AWS CLI是否正常工作。

### 4.2.2 配置CLI

要使用CLI，需要验证用户的身份。目前为止，我们使用了AWS根账号。这个账号能做任何事，这是好事也是坏事。强烈建议读者不要使用AWS根账号（第6章将介绍更多安全相关的知识），所以我们要创建一个新用户。

打开AWS管理控制台，在导航栏中点击“服务”，然后点击“IAM服务”，会显示图4-4所示的页面，选择左边的“用户”。

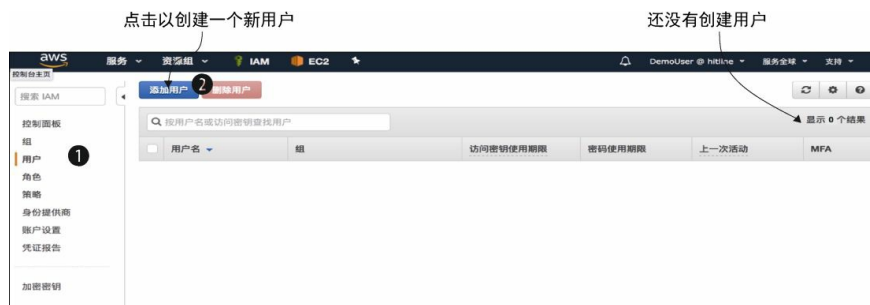


图4-4 IAM用户（空）



图4-5 创建一个IAM用户

按下面的步骤创建一个新用户。

- （1）点击“添加用户”，打开如图4-5所示的页面。
- （2）输入mycli作为第一个用户的用户名。
- （3）让其他项保持空白，选择“编程访问”。点击“下一步：权限”按钮。
- （4）直接点击“下一步：审核”，然后点击“创建用户”按钮。

跳过其余步骤，将会打开如图4-6所示的页面。点击“显示”来显示私有访问密钥——它只会显示一次！现在用户需要复制这一凭证到CLI

配置中。接下来学习它是怎么工作的。



图4-6 创建一个IAM用户：显示密钥凭证

在用户的计算机上打开终端（Windows上的PowerShell或OS X和Linux上的Bash shell，不是AWS管理控制台），然后运行`aws configure`。用户会被问及4个信息。

- AWS访问密钥ID ——从访问密钥ID框（浏览器窗口）中复制这一值。
- AWS私有访问密钥 ——从私有访问密钥框（浏览器窗口）中复制这一值。
- 默认区域名称 ——输入`us-east-1`。
- 默认输出格式 ——输入`json`。

最后，在终端上应该看上去像这样：

```
$ aws configure
AWS Access Key ID [None]: AKIAJXMDAVKCM5ZTX7PQ
AWS Secret Access Key [None]: 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
Default region name [None]: us-east-1
Default output format [None]: json
```

现在CLI被配置好了，使用用户mycli进行身份认证。切换回浏览器窗口然后点击“关闭”，结束用户创建向导，将会打开图4-7所示的页面。



图4-7 IAM用户

接下来就需要处理授权来确定允许用户mycli做些什么。目前，这个用户不被允许做任何事（默认设定）。点击用户mycli，会看到图4-8所示的页面。



图4-8 没有任何权限的IAM用户

在“权限”部分，点击“添加权限”按钮，将打开图4-9所示的页面，选择“直接附加现有策略”。



图4-9 向一个IAM用户关联一个策略

搜索Admin，然后选择策略AdministratorAccess。点击“下一步：审核”，然后点击“添加权限”。现在用户mycli看上去如图4-10所示。

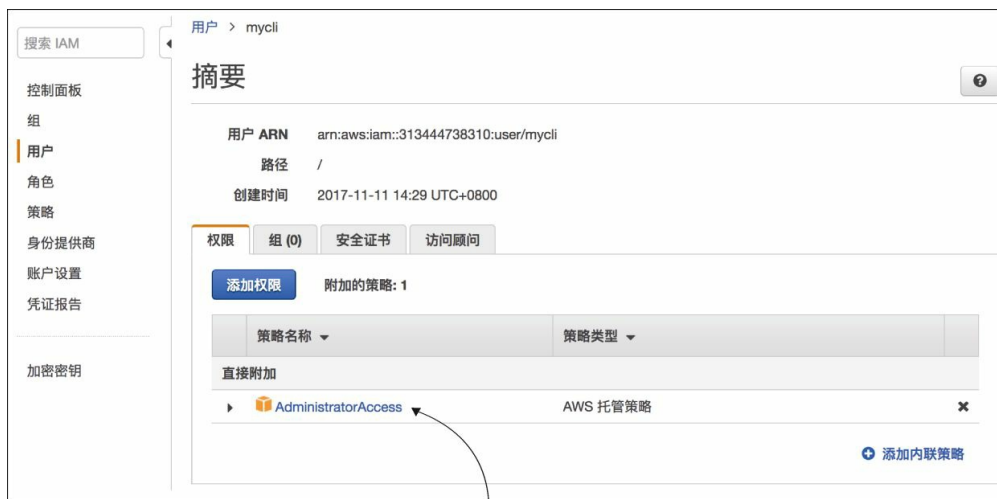


图4-10 拥有admin权限的IAM用户mycli

现在可以测试CLI是否工作了。切换到终端窗口，输入aws ec2 describe-regions 来获取所有可用区域的列表：



```
$ aws ec2 describe-regions
{
  "Regions": [
    {
      "Endpoint": "ec2.eu-central-1.amazonaws.com",
      "RegionName": "eu-central-1"
    },
    {
      "Endpoint": "ec2.sa-east-1.amazonaws.com",
      "RegionName": "sa-east-1"
    },
    [...]
    {
      "Endpoint": "ec2.ap-southeast-2.amazonaws.com",
      "RegionName": "ap-southeast-2"
    },
    {
      "Endpoint": "ec2.ap-southeast-1.amazonaws.com",
      "RegionName": "ap-southeast-1"
    }
  ]
}
```

成功了！现在可以开始使用CLI了。

### 4.2.3 使用CLI

假如用户想要获得所有类型为t2.micro的EC2实例的列表。在终端上执行**aws**，如下所示：

```
$ aws ec2 describe-instances --filters "Name=instance-type,Values=t2.micro"
{
  "Reservations": []      <--空列表，因为还没有创建EC2 实例
}
```

要使用AWS CLI，需要指定一个服务与操作。在上一个例子中，服务是**ec2**，操作是**describe-instances**，可以添加选项**--key value**

:

```
$ aws <service> <action> [--key value ...]
```

CLI的一个重要的特色是**help** 关键字，使用这个关键字可以得到3个级别的详细帮助信息。

- **aws help** ——显示所有可用的服务。
- **aws <service> help** ——显示某一服务所有可用的操作。
- **aws <service> <action> help** ——显示特定服务操作可用的所有选项。

有时候用户需要临时的计算力，如Linux服务器要通过SSH做测试。要做到这一点，可以编写一个脚本来创建一台虚拟服务器。这个脚本将运行在用户的本地计算机上，并输出用户是如何通过SSH连接到服务器的。等用户完成自己的测试，脚本将能够终止这台虚拟服务器。这个脚本的使用如下：

```
$ ./server.sh
waiting for i-c033f117 ...      <-- 等到启动
i-c033f117 is accepting SSH connections under ec2-54-164-72-62 ...
ssh -i mykey.pem ec2-user@ec2-54-[...]aws.com      <-- SSH 连接字符串
Press [Enter] key to terminate i-c033f117 ...
[...]
terminating i-c033f117 ...      <-- 等到终止
done.
```

用户的服务器一直运行直到按下Enter键。当用户按下Enter键时，服务器将被终止。

这个方案的局限性如下。

- 同一时刻只能处理一台服务器。
- Windows有一个与Linux和Mac OS X不同的版本。
- 它是一个命令行应用，而不是图形化应用。

然而，这个CLI方案解决了系列使用场景。

- 创建一台虚拟服务器。
- 获取虚拟服务器的公有DNS名用于SSH连接。
- 当不再需要时，终止一台虚拟服务器。

根据用户所用的操作系统不同，可以使用Bash（Linux和Mac OS X）或PowerShell（Windows）来写脚本。

在开始之前需要解释一个CLI的重要功能。`--query` 选项使用 JMESPath（一种JSON的查询语言）从结果中提取数据。这是非常有用的，因为通常用户只需要结果中某个特别的项。让我们来看看下面的JSON中JMESPath的操作。这是`aws ec2 describe-images`的结果，列出了可用的AMI。要启动一个EC2实例，需要ImageId，使用 JMESPath可以提取到这一信息：

```
{
  "Images": [
    {
      "ImageId": "ami-146e2a7c",
      "State": "available"
    },
    {
      "ImageId": "ami-b66ed3de",
      "State": "available"
    }
  ]
}
```

要提取第一个ImageId，路径为`Images[0].ImageId`，这个查询的结果是`"ami-146e2a7c"`。要提取所有State，路径为`Images[*].State`，这个查询的结果为`["available", "available"]`。使用JMESPath的简单介绍，我们已经能够提取到所需的数据。

Linux和Mac OS X能解释脚本，而Windows更喜欢PowerShell脚本。我们创建了同一脚本的两个版本。

## 1. Linux和Mac OS X

读者能在本书的代码目录中的/`chapter4/server.sh` 找到代码清单4-3，可以复制并粘贴每一行到终端或通过`chmod +x server.sh && ./server.sh` 执行整个脚本。

代码清单4-3 使用CLI（Bash）创建与终止一台服务器

```
#!/bin/bash -e      <---当命令出错时中止
AMIID=$(aws ec2 describe-images --filters "Name=description, \      <---获取
Amazon Linux AMI的ID

Values=Amazon Linux AMI 2015.03.? x86_64 HVM GP2" \
--query "Images[0].ImageId" --output text)

VPCID=$(aws ec2 describe-vpcs --filter "Name=isDefault, Values=true" \
    <---获取默认VPC 的ID
--query "Vpcs[0].VpcId" --output text)

SUBNETID=$(aws ec2 describe-subnets --filters "Name=vpc-id, Values=$VPCID"
\
--query "Subnets[0].SubnetId" --output text)      <---获取默认子网ID

SGID=$(aws ec2 create-security-group --group-name mysecuritygroup \      <--
-创建安全组
--description "My security group" --vpc-id $VPCID --output text)

aws ec2 authorize-security-group-ingress --group-id $SGID \      <---允许入站
SH 连接
--protocol tcp --port 22 --cidr 0.0.0.0/0

INSTANCEID=$(aws ec2 run-instances --image-id $AMIID --key-name mykey \
    <---创建并启动服务器
--instance-type t2.micro --security-group-ids $SGID \
--subnet-id $SUBNETID --query "Instances[0].InstanceId" --output text)

echo "waiting for $INSTANCEID ..."

aws ec2 wait instance-running --instance-ids $INSTANCEID      <---等到服务器
启动

PUBLICNAME=$(aws ec2 describe-instances --instance-ids $INSTANCEID \      <
--获取服务器的公有域名
--query "Reservations[0].Instances[0].PublicDnsName" --output text)
```

```

echo "$INSTANCEID is accepting SSH connections under $PUBLICNAME"
echo "ssh -i mykey.pem ec2-user@$PUBLICNAME"
read -p "Press [Enter] key to terminate $INSTANCEID ..."
aws ec2 terminate-instances --instance-ids $INSTANCEID      <-- 终止服务器
    echo "terminating $INSTANCEID ..."
aws ec2 wait instance-terminated --instance-ids $INSTANCEID  <-- 等到服务器终止
aws ec2 delete-security-group --group-id $SGID      <-- 删除安全组

```

### 资源清理

在继续下一步操作之前一定要确保已经终止了服务器！

## 2. Windows

代码清单4-4可以在本书的代码目录/`chapter4/server.ps1` 中找到。

右键点击`server.ps1`文件，选择Run with PowerShell来执行这一脚本。

代码清单4-4 用CLI（PowerShell）来创建和终止一台服务器

```

$ErrorActionPreference = "Stop"<-- 当命令出错时中止

$AMIID=aws ec2 describe-images --filters "Name=description, \      <-- 获取Amazon Linux AMI 的ID
Values=Amazon Linux AMI 2015.03.? x86_64 HVM GP2" \
--query "Images[0].ImageId" --output text

$VPCID=aws ec2 describe-vpcs --filter "Name=isDefault, Values=true" \
<-- 获取默认VPC 的ID
--query "Vpcs[0].VpcId" --output text

$SUBNETID=aws ec2 describe-subnets --filters "Name=vpc-id, Values=$VPCID" \
\
--query "Subnets[0].SubnetId" --output text      <-- 获取默认子网ID

$SGID=aws ec2 create-security-group --group-name mysecuritygroup \      <-- 创建安全组

```

```

--description "My security group" --vpc-id $VPCID \
--output text

aws ec2 authorize-security-group-ingress --group-id $SGID \      <--- 允许入站
SSH 连接
--protocol tcp --port 22 --cidr 0.0.0.0/0

$INSTANCEID=aws ec2 run-instances --image-id $AMIID --key-name mykey \
  <--- 创建并启动服务器
--instance-type t2.micro --security-group-ids $SGID \
--subnet-id $SUBNETID \
--query "Instances[0].InstanceId" --output text

Write-Host "waiting for $INSTANCEID ..."
aws ec2 wait instance-running --instance-ids $INSTANCEID      <--- 等到服务器
启动

$PUBLICNAME=aws ec2 describe-instances --instance-ids $INSTANCEID \      <--
- 获取服务器的公有域名
--query "Reservations[0].Instances[0].PublicDnsName" --output text

Write-Host "$INSTANCEID is accepting SSH under $PUBLICNAME"
Write-Host "connect to $PUBLICNAME via SSH as user ec2-user"
Write-Host "Press [Enter] key to terminate $INSTANCEID ..."
Read-Host
aws ec2 terminate-instances --instance-ids $INSTANCEID      <--- 终止服务器
Write-Host "terminating $INSTANCEID ..."
aws ec2 wait instance-terminated --instance-ids $INSTANCEID      <--- 等到服务
器终止
aws ec2 delete-security-group --group-id $SGID      <--- 删除安全组

```

#### 资源清理

在继续下一步操作之前一定要确保已经终止了服务器！

### 3. 为什么要写脚本

为什么要写脚本，而不是使用图形化的AWS管理控制台？脚本可以复用，并且在长时间操作时节省时间。用户能够使用自己之前的项目中已经可用的模块来快速创建新的架构。通过使自己的基础架构创建自

动化，用户也能够加强自己的部署流水线的自动化。

另一个好处是，脚本将是能想象的最准确的文档（甚至计算机能理解它）。如果你想在周一重复上周五自己做的事，脚本就是无价之宝。如果你病了，而你的同事需要处理好你的任务，他们会感激你留下了脚本。

## 4.3 使用SDK编程

AWS为许多编程语言提供软件开发套件（Software Development Kits，SDK）：

- Android
- Python
- Node.js（JavaScript）
- Java
- Browsers（JavaScript）
- Ruby
- PHP
- .NET
- iOS
- Go

AWS SDK是从用户喜欢的编程语言调用AWS API的便捷方法。SDK会处理好类似认证、重试、HTTPS通信和JSON（还原）序列化。用户可以选择自己喜欢的语言的SDK，但是，在本书中所有示例使用JavaScript，并且在Node.js运行环境中运行。

### 安装并开始使用Node.js

Node.js是一个在事件驱动环境下执行JavaScript的平台，且容易创建网络应用。要安装Node.js，访问Node.js官方网站，然后下载适合自己的操作系统的程序包。

Node.js安装好后，就可以通过在终端输入`node --version`来验证一切都可用。终端应该会产生类似于v0.12.\*的回应。现在就可以运行我们的JavaScript示例了，如Node Control Center for AWS。

随着Node.js安装的有一个重要的工具叫作npm，它是Node.js的程序包管理器。在终端上运行`npm --version`来验证安装。

要在Node.js中运行JavaScript脚本，在终端输入`node script.js`。本书中使用Node.js是因为它容易安装，不需要IDE，并且大多数程序员是熟悉其语法的。

不要混淆术语JavaScript和Node.js。如果想弄精确些，JavaScript是编程语言，而Node.js是执行环境。但是，别期待任何人能做这样的区分。Node.js也叫作node。

为了理解Node.js（JavaScript）上的AWS SDK是如何工作的，让我



们创建一个Node.js（JavaScript）应用来通过AWS SDK控制EC2服务器。

### 4.3.1 使用SDK控制虚拟服务器：nodecc

Node Control Center for AWS（nodecc）是一个用JavaScript编写的有文本UI的能管理多个临时EC2服务器的应用。nodecc具有下列功能。

- 能处理多个服务器。
- 用JavaScript编写且运行在Node.js上，因此它能跨平台使用。
- 使用文本UI。

图4-11展示了nodecc的开始界面。



图4-11 Node Control Center for AWS：开始界面

读者可以在本书的代码目录/`chapter4/nodecc/` 中找到nodecc应用。切换到那个目录，在终端上运行`npm install`来安装所有需要的依赖项。要启动nodecc，先运行`node index.js`。总是可以使用左箭头键来返回，按Esc键或q键来退出应用。

SDK使用你为CLI所创建的相同设置，所以当你运行nodecc时也使用用户mycli。

### 4.3.2 nodecc如何创建一台服务器

在能使用nodecc做任何事之前，用户需要至少一台服务器。要启动一台服务器，选择AMI，如图4-12所示。

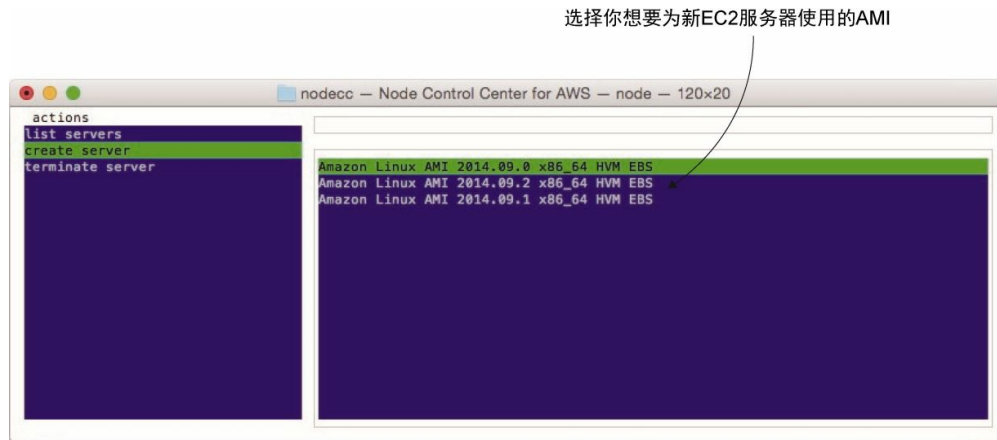


图4-12 nodecc: 创建一台服务器（2步中的第一步）

用于获取可用AMI列表的代码在lib/listAMIs.js中，具体内容如代码清单4-5所示。

代码清单4-5 /lib/listAMIs.js

```
var jmespath = require('jmespath');    <--require 用来装载模块
var AWS = require('aws-sdk');

var ec2 = new AWS.EC2({"region": "us-east-1"});    <--配置一个EC2 端点

module.exports = function(cb) {    <--module.exports 使这个函数能被listAMI
  模块的用户使用
  ec2.describeImages({    <--操作
    "Filters": [{
      "Name": "description",
      "Values": ["Amazon Linux AMI 2015.03.? x86_64 HVM GP2"]
    }]
  }, function(err, data) {
    if (err) {    <--万一出错，设置错误
      cb(err);
    } else {    <--否则，data 中包含所有AMI
      var amiIds = jmespath.search(data, 'Images[*].ImageId');
      var descriptions = jmespath.search(data, 'Images[*].Description');
      cb(null, {"amiIds": amiIds, "descriptions": descriptions});
    }
  });
};
```

这段代码是这样结构化的，每个操作都在lib库文件夹中实现。创建一台服务器的下一步是选择服务器启动时应该在的子网。我们还没有学习子网，因此目前先随机选择一个，如图4-13所示。相应的脚本位于lib/listSubnets.js。

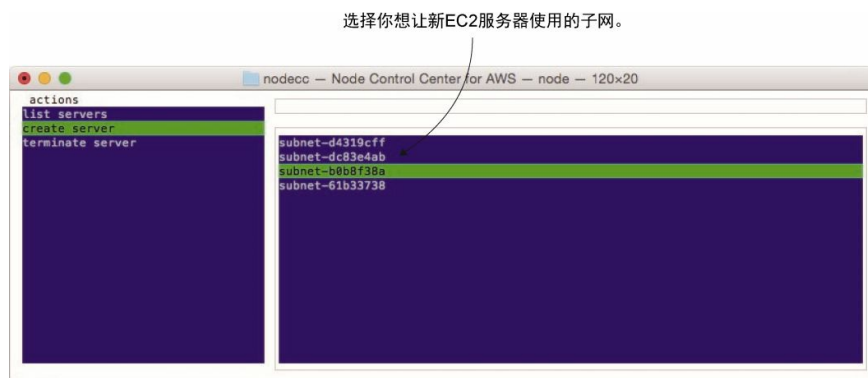


图4-13 nodecc：创建一台服务器（2步的第二步）

在选择了子网之后，服务器将由lib/createServer.js创建，然后我们会看见一个启动屏幕。现在是时候找出新创建的服务器的公有DNS名了。使用左箭头键切换到导航部分。

### 4.3.3 nodecc是如何列出服务器并显示服务器的详细信息

一个重要的使用场景是nodecc必须支持显示能通过SSH连接的服务器的公有名。因为nodecc处理多台服务器，第一步是选择一台服务器，如图4-14所示。

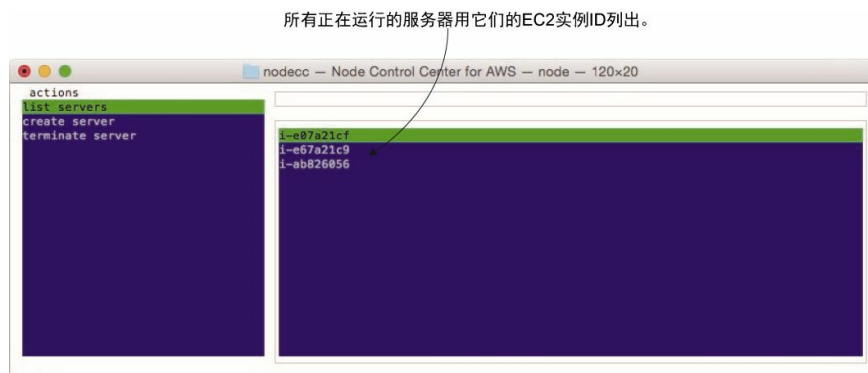


图4-14 nodecc: 列出服务器

让我们看一下`lib/listServers.js` 中是如何使用AWS SDK来获取服务器列表的。在选择服务器之后，就可以显示它的详细信息，如图4-15所示。你可以通过SSH使用`PublicDnsName` 连接到这台服务器实例。按左箭头键切换回导航部分。

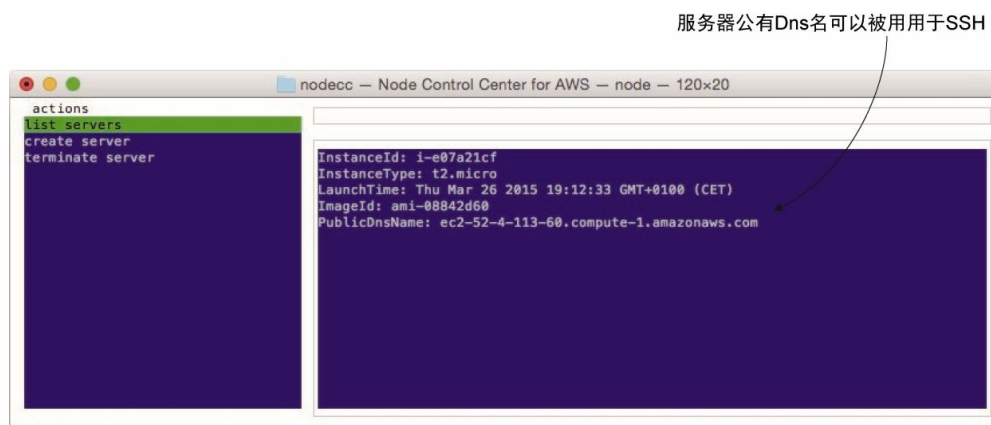


图4-15 nodecc: 显示服务器的详细信息

### 4.3.4 nodecc如何终止一台服务器

用户首先需要选择服务器以进行终止操作。再次使用`lib/listServers.js` 列出服务器。选择好要终止的服务器后，使用`lib/terminateServer.js` 会负责进行终止。

这就是nodecc——一个用于控制临时的EC2服务器的文本界面程序。读者可以花些时间想想看，使用自己最喜欢的编程语言和AWS

SDK可以搭建什么应用，很有可能会想到一个不错的商机。

#### 资源清理

在继续下一步操作之前一定要确保已经终止了所有的服务器！

## 4.4 使用蓝图来启动一台虚拟服务器

早些时候，我们谈到JIML来引入基础架构即代码的概念。幸运的是，AWS已经提供了一个比JIML更好的工具：**AWS CloudFormation**。CloudFormation基于模板来创建基础架构，也就是我们称作蓝图的东西。

### 注意

通常我们在讨论基础架构自动化时使用蓝图（blueprint）这样的术语。配置管理服务AWS CloudFormation使用的蓝图被称为模板（template）。

模板使用JSON描述了用户的基础架构，CloudFormation能对其进行解析。用户只需要使用声明的表述语言，而不是给出实现它需要的所有操作。描述的方法意味着用户告诉CloudFormation需要什么样的基础架构以及组件之间是怎样连接的。用户不需要告诉CloudFormation需要哪些操作来创建那样的基础架构，不需要定义操作被执行的顺序。再次强调，处理组件之间的依赖关系很重，但是CloudFormation还能提供更多的好处。

CloudFormation的好处如下。

- 它让用户在AWS平台上统一的方式来描述基础架构。如果用户用脚本来创建自己的基础架构，每个人会用不同的方法解决同样的问题。这对新开发人员与运维人员是个障碍，他们需要努力去理解代码要做什么。CloudFormation模板是一个定义基础架构的清晰的语言。
- 它能处理依赖关系。试过把网络服务器注册到一个还不可用的负载均衡器吗？初看起来，使用脚本的方法用户会搞错很多依赖项。相信我们：永远不要尝试使用脚本来创建一个复杂的基础架构。组件之间的依赖关系会变得一团糟！
- 它是可复制的。如何使用用户的测试环境和生产环境保持完全一致？使用CloudFormation，用户能创建两个完全一样的基础架构并且保持它们同步。
- 它是可自定义的。用户可以向CloudFormation插入自定义的参数来

按期望自定义模板。

- 它是可测试的。从模板创建基础架构是可测试的。随时可以按需启动一个新的基础架构，运行测试，完成后再关掉它。
- 它是可更新的。CloudFormation可以更新用户的基础架构。它将找出模板中改变了的部分，然后将这些变化尽可能平滑地应用到现有的基础架构。
- 它最小化人为的误操作。CloudFormation不会感到疲倦——即使是在凌晨3点。
- 它把基础架构文档化。CloudFormation模板是一个JSON文档。用户可以把它当作代码，然后使用一个版本控制系统（如Git）来跟踪变更。
- 它是免费的。CloudFormation服务本身不会产生额外费用。

我们认为CloudFormation是在AWS上管理基础架构的最强的工具之一。

## 4.4.1 CloudFormation模板解析

一个基本的CloudFormation模板分为5个部分。

（1）格式版本 ——最新的模板格式版本是2010-09-09，且是目前唯一合法的值。指定这个值，默认是最新版本，如果将来引入新格式的模板，这有可能会引发问题。

（2）描述 ——这个模板是关于什么的？

（3）参数 ——参数使用值用来自定义模板。例如，域名、客户ID和数据库密码。

（4）资源 ——一项资源是用户能描述的最小组件。例如，虚拟服务器、负载均衡器或弹性IP地址。

（5）输出 ——输出和参数有点儿像，但是用途正好相反。输出从模板返回一些信息，如一台EC2服务器的公有域名。

一个基本模板如代码清单4-6所示。

```
{
  "AWSTemplateFormatVersion": "2010-09-09",      <--唯一合法版本
  "Description": "CloudFormation template structure",  <--这个模板是关于
什么的
  "Parameters": {
    [...]      <--定义参数
  },
  "Resources": {
    [...]      <--定义资源
  },
  "Outputs": {
    [...]      <--定义输出
  }
}
```

让我们进一步看看参数、资源和输出。

## 1. 格式版本及描述

唯一合法的**AWSTemplateFormatVersion** 值目前是**"2010-09-09"**。用户需要指定格式版本。如果不指定，CloudFormation会认为是最新版本。前面提到，这意味着如果在将来有了一个新的格式版本，会陷入严重的麻烦之中。

**Description** 不是强制的，但是建议读者花些时间来描述模板的用途。一个有意义的描述将来有助于自己记起这个模板是干什么的，它也能帮助其他同事理解。

## 2. 参数

参数至少有一个名字和类型。建议用户同时添加一个描述，如代码清单4-7所示。

```
{
```



```
[...]
"Parameters": {
  "NameOfParameter": {      <---参数名
    "Type": "Number",        <---这个参数是个数字
    "Description": "This parameter is for demonstration"
  },
  [...]
},
[...]
```

表4-1列出了合法的类型。

表4-1 CloudFormation参数类型

类 型	描 述
String CommaDelimitedList	一个字符串或由逗号分隔的字符串列表
Number List<Number>	一个整数或浮点数或整数列表或浮点数列表
AWS::EC2::Instance::Id List<AWS::EC2::Instance::Id>	一个EC2实例ID（虚拟服务器）或一个EC2实例ID列表
AWS::EC2::Image::Id List<AWS::EC2::Image::Id>	一个AMI ID或AMI列表
AWS::EC2::KeyPair::KeyName	一个Amazon EC2密钥对名
AWS::EC2::SecurityGroup::Id List<AWS::EC2::SecurityGroup::Id>	一个安全组ID或安全组ID列表
AWS::EC2::Subnet::Id List<AWS::EC2::Subnet::Id>	一个子网ID或子网ID列表

AWS::EC2::Volume::Id List<AWS::EC2::Volume::Id>	一个EBS卷ID（网络附加存储）或EBS卷ID列表
AWS::EC2::VPC::Id List<AWS::EC2::VPC::Id>	一个VPCID（虚拟私有网络）或VPC ID列表
AWS::Route53::HostedZone::Id List<AWS::Route53::HostedZone::Id>	一个DNS区域ID或DNS区域ID列表

除了使用Type 与Description，用户还可以使用表4-2中列出的属性来增强一个参数。

表4-2 CloudFormation参数属性

属 性	描 述	例 子
Default	参数的默认值	
NoEcho	在所有图形化工具中隐藏参数值（对密码有用）	"NoEcho": true
AllowedValues	指定参数的可能值	"AllowedValues": ["1", "2", "3"]
AllowedPattern	比AllowedValues更通用，因为它使用正则表达式	"AllowedPattern": "[a-zA-Z0-9]*" 只允许a~z、A~Z和0~9，长度任意
MinLength、MaxLength	与字符串类型一起使用，用来定义最小长度和最大长度	
MinValue、MaxValue	与数字类型一起使用，用来定义上下限	

CloudFormation模板的参数部分如下：

```

{
  [...]
  "Parameters": {
    "KeyName": {
      "Description": "Key Pair name",
      "Type": "AWS::EC2::KeyPair::KeyName"      <--- 只允许KeyName
    },
    "NumberOfServers": {
      "Description": "How many servers do you like?",
      "Type": "Number",
      "Default": "1",      <---默认为一台服务器
      "MinValue": "1",
      "MaxValue": "5"      <---设置上限以免产生大量开销
    },
    "WordPressVersion": {
      "Description": "Which version of WordPress do you want?",
      "Type": "String",
      "AllowedValues": ["4.1.1", "4.0.1"]      <---限制特定版本
    }
  },
  [...]
}

```

现在我们应该对参数有了更好的感觉。如果想了解参数的一切，可以访问AWS官方网站或紧跟本书内容动手学习。

### 3. 资源

一个资源至少有一个名字、一个类型和一些属性，如代码清单4-8所示。

代码清单4-8 CloudFormation资源结构

```

{
  [...]
  "Resources": {
    "NameOfResource": {      <---参数名
      "Type": "AWS::EC2::Instance",      <---定义一台EC2 服务器
      "Properties": {
        [...]      <---资源类型所需的属性
      }
    }
  }
}

```

```
    }  
  },  
  [...]  
}
```

定义资源时，用户需要知道类型和该类型所需的属性。在本书中，读者将了解许多资源类型以及它们各自的属性。代码清单4-9展示了单台EC2服务器的一个例子。如果看见{"Ref": "NameOfSomething"}，把它当作一个占位符，应替换为名称的引用。用户可以引用参数和资源来创建依赖关系。

代码清单4-9 CloudFormation EC2服务器资源

```
{  
  [...]  
  "Resources": {  
    "Server": {      <--资源名  
      "Type": "AWS::EC2::Instance",    <--定义一台EC2 服务器  
      "Properties": {  
        "ImageId": "ami-1ecae776",    <--一些硬编码的设置  
        "InstanceType": "t2.micro",  
        "KeyName": {"Ref": "KeyName"},    <--这些设置通过参数定义  
        "SubnetId": {"Ref": "Subnet"}  
      }  
    }  
  },  
  [...]  
}
```

现在我们描述了服务器，但如何输出它的公有DNS名呢？

## 4. 输出

CloudFormation模板的输出包括至少一个名称（如参数和资源）和一个值，建议读者同时添加一个描述。读者可以使用输出来将数据从自己的模板传递到外面（见代码清单4-10）。

代码清单4-10 CloudFormation输出结构

```

{
  [...]
  "Outputs": {
    "NameOfOutput": {    <---输出的名称
      "Value": "1",      <---输出的值
      "Description": "This output is always 1"
    }
  }
}

```

静态输出不是很有用。大多数时候，用户会引用资源的名称或资源的一个属性，如它的公有DNS名，如代码清单4-11所示。

代码清单4-11 CloudFormation输出示例

```

{
  [...]
  "Outputs": {
    "ServerEC2ID": {
      "Value": {"Ref": "Server"},    <---引用EC2 服务器
      "Description": "EC2 ID of the server"
    },
    "PublicName": {
      "Value": {"Fn::GetAtt": ["Server", "PublicDnsName"]},    <---获得EC2 服务器的属性公有DNS 名
      "Description": "Public name of the server"
    }
  }
}

```

本书稍后会介绍**Fn::GetAtt**的一些最重要的属性。如果了解所有的属性，可访问AWS官方网站。

现在让我们简单看一下CloudFormation模板的核心部分，是时候来制作一个自己的模板了。

## 4.4.2 创建第一个模板

假设你需要为开发团队提供一台虚拟服务器。几个月之后，开发团队意识基于应用需求的变化，这台虚拟服务器需要更多的CPU。你可以使用CLI和SDK处理这一要求，但是正如3.4节所介绍的，在更改实例类型前，用户必须先停止实例。具体流程如下：停止实例，等待实例停止，更改实例类型，启动实例，等待实例启动。

CloudFormation使用的描述法更简单：只需改变**InstanceType** 属性，然后更新模板。**InstanceType** 可以通过参数传给模板。就是这样！你可以开始创建模板了，如代码清单4-12所示。

代码清单4-12 用CloudFormation模板创建一个EC2实例

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "AWS in Action: chapter 4",
  "Parameters": {
    "KeyName": {      <--用户定义将使用哪个密钥
      "Description": "Key Pair name",
      "Type": "AWS::EC2::KeyPair::KeyName",
      "Default": "mykey"
    },
    "VPC": {          <--6.5 节将介绍这一内容
      [...]
    },
    "Subnet": {       <--6.5 节将介绍这一内容
      [...]
    },
    "InstanceType": { <--用户定义实例类型
      "Description": "Select one of the possible instance types",
      "Type": "String",
      "Default": "t2.micro",
      "AllowedValues": ["t2.micro", "t2.small", "t2.medium"]
    }
  },
  "Resources": {
    "SecurityGroup": { <--6.4 节将介绍这一内容
      "Type": "AWS::EC2::SecurityGroup",
      "Properties": {
        [...]
      }
    },
    "Server": {        <--定义最小EC2 实例
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "ImageId": "ami-1ecae776",
```

```

        "InstanceType": {"Ref": "InstanceType"},
        "KeyName": {"Ref": "KeyName"},
        "SecurityGroupIds": [{"Ref": "SecurityGroup"}],
        "SubnetId": {"Ref": "Subnet"}
    }
}
},
"Outputs": {    <-- 返回EC2 实例的公有DNS 名
    "PublicName": {
        "Value": {"Fn::GetAtt": ["Server", "PublicDnsName"]},
        "Description": "Public name (connect via SSH as user ec2-user)"
    }
}
}
}

```

读者可以在本书的代码目录/`chapter4/server.json` 中找到这个模板的完整代码。目前不要担心VPC、子网和安全组，这些内容在第6章中会详细介绍。

#### 模板在哪里

这个模板可以从下载的源代码中找到。我们谈到的文件位于`chapter4/server.json`。在S3上，相同的文件位于<https://s3.amazonaws.com/awsinaction/chapter4/server.json>。

如果从模板创建一个基础架构，则CloudFormation称之为堆栈。可以认为模板对应堆栈，就像是类对应对象。模板只存在一次，而许多堆栈可以从同一个模板中被创建。

打开AWS管理控制台。在导航栏中点击“服务”，然后点击CloudFormation服务。图4-16显示了初始CloudFormation界面，显示了所有堆栈的概览。



图4-16 CloudFormation堆栈概览

下面的步骤将引导用户创建自己的堆栈。

- (1) 点击“创建新堆栈”按钮启动一个4步的向导。
- (2) 给堆栈起名，如**server1**。
- (3) 选择“Specify an Amazon S3 template URL”，然后输入 [https://s3.amazonaws.com/ awsinaction/chapter4/server.json](https://s3.amazonaws.com/awsinaction/chapter4/server.json)，如图4-17所示。

在第二步中，定义下面的参数。

- (1) **InstanceType**：选择**t2.micro**。
- (2) **KeyName**：选择**mykey**。
- (3) **Subnet**：选择下拉列表中的第一个值。子网稍后会介绍。
- (4) **VPC**：选择下拉列表中的第一个值。**VPC**稍后会介绍。



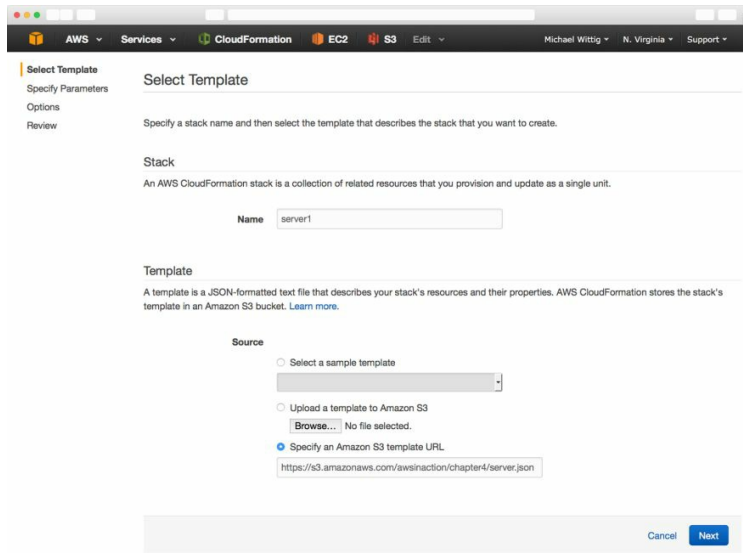


图4-17 创建一个CloudFormation堆栈：选择一个模板（4步的第一步）

图4-18展示了参数设置步骤。在为每个参数选择了值之后点击“下一步”。



图4-18 创建一个CloudFormation堆栈：定义参数（4步的第二步）

在第三步，可以为堆栈定义标签。所有由堆栈创建的资源都会被自动打上这些标签。创建一个新标签，输入**system**作为键值，**tempserver**作为值。点击“下一步”。第四步显示这个堆栈的总结，如图4-19所示。

创建堆栈

选择模板

指定详细信息

选项

审核

审核

模板

模板 URL

https://s3.amazonaws.com/awsinaction/chapter4/server.json

描述

AWS in Action: chapter 4

估算费用

费用

详细信息

堆栈名称

server1

InstanceType

t2.micro

KeyName

mykey

Subnet

subnet-148a9f52

VPC

vpc-2615e443

选项

标签

没有提供标签

高级

通知

终止保护

禁用

超时

无

失败时回滚

是

取消

上一步

创建

图4-19 创建一个CloudFormation堆栈：总结（4步的第四步）

点击“创建”。现在堆栈创建好了。如果这个过程成功，会看见如图4-20所示的界面。只要“状态”还是CREATE\_IN\_PROGRESS，就需要耐心等待。当“状态”变为CREATE\_COMPLETE，选择这个堆栈，点击“输出”标签页来查看服务器的公有域名。

CloudFormation		堆栈	
创建堆栈	操作	设计模板	
筛选条件: 活跃	按堆栈名称	正在显示 1 个 堆栈	
堆栈名称	创建时间	状态	描述
<input checked="" type="checkbox"/> server1	2017-11-11 14:56:19 UTC+0800	CREATE_COMPLETE	AWS in Action: chapter 4
<div>概述 输出 资源 事件 模板 参数 标签 堆栈策略 更改集</div>			
键	值	描述	导出名称
PublicName	ec2-13-114-123-39.ap-northeast-1.com pute.amazonaws.com	Public name (connect via SSH as user...	

图4-20 创建CloudFormation堆栈

是时候来测试新功能了——修改实例类型。选择这个堆栈，选择“操作”菜单，点击“更新堆栈”菜单项。启动的向导和创建堆栈时的操作类似。图4-21显示了向导的第一步。



图4-21 更新CloudFormation堆栈：总结（4步的第一步）

确认选中了“使用当前模板”。在第二步，需要更改InstanceType，选择t2.small或t2.medium来让服务器计算能力翻两番。

#### 警告

启动一台实例类型为t2.small或t2.medium的虚拟服务器会产生费用。

第三步是关于更新堆栈时的复杂选项。现在还不需要这些功能，所以点击“下一步”跳过这一步。第四步是一个总结：点击“更新”。现在堆栈的“状态”变为UPDATE\_IN\_PROGRESS。几分钟后，“状态”应该会变为UPDATE\_COMPLETE。可以选择这个堆栈，通过点击“输出”标签页来查看服务器的公有域名。

#### CloudFormation的替代方案

如果用户不想写JSON文本来为自己的基础架构创建模板，有几个替代CloudFormation的方案。像Troposphere（一个用Python写的库）这样的工具可以帮助用户创建CloudFormation模板，而不需要写JSON。它们在CloudFormation上另外加一个抽象层来实现这一点。

还有一些工具能让用户使用基础架构即代码，而不需要CloudFormation，如Terraform和Ansible让用户将自己的基础架构描述为代码。

修改参数时，CloudFormation会找出需要做些什么才能达到最终结

果。这就是描述法的力量：说出最终结果是什么样，而不是怎样达到最终结果。

#### 资源清理

选择堆栈并且选择“操作”菜单，点击“删除堆栈”菜单项。

## 4.5 小结

- 可以使用命令行接口（CLI）、开发工具套件或CloudFormation在AWS上自动化自己的基础架构。
- 基础架构即代码描述了编写程序来创建与修改基础架构（包括虚拟服务器、网络、存储等）的方法。
- 可以使用CLI通过脚本（Bash与PowerShell）的方式在AWS上自动化复杂的流程。
- 可以使用9种编程语言的SDK来将AWS嵌入自己的应用并创建nodecc这样的应用。
- CloudFormation使用JSON描述法：用户只需要定义自己的基础架构的最终状态，而CloudFormation会找出如何达到这个状态。  
CloudFormation模板的主要部分有参数、资源和输出。

## 第5章 自动化部署：CloudFormation、Elastic Beanstalk和OpsWorks

### 本章主要内容

- 在服务器启动时运行脚本来部署应用
- 在AWS Elastic Beanstalk的帮助下部署普通的网站应用
- 在AWS OpsWorks的帮助下部署多层应用
- 比较AWS上的部署服务的不同

不论想用自主开发的、开源项目的，还是商业厂商的软件，都需要安装、更新和配置应用程序及其依赖的组件。这一过程称为部署。在本章中，我们将学习AWS上用于部署应用的3个工具。

- 使用AWS CloudFormation并在引导结束时启动一个脚本来部署一个VPN方案。
- 使用AWS Elastic Beanstalk来部署一个协作文本编辑器。文本编辑器Etherpad是一个简单的网络应用程序，非常适合使用AWS Elastic Beanstalk进行部署，因为这个服务原生支持Node.js平台。
- 使用AWS OpsWorks部署一个IRC网络客户端和IRC服务器。安装包含两部分：一个用于分发IRC网络客户端的Node.js服务器和IRC服务器本身。这个例子包含了多层结构，非常适合AWS OpsWorks。

虽然本章中所选的示例都不需要存储方案，但是这3个部署方案都支持有存储方案的应用的发布。读者可以在本书的下一部分找到使用存储的例子。

#### 示例都包含在免费套餐中

本章中的所有示例都包含在免费套餐中。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

在服务器上部署一个典型的网站应用必需的步骤有哪些呢？下面以一个广泛使用的博客平台WordPress为例加以说明。

（1）安装Apache HTTP服务器、MySQL数据库、PHP运行环境、供PHP调用的MySQL访问库和一个SMTP邮件服务器。

（2）下载WordPress应用，然后在服务器上解压缩。

（3）配置Apache网站服务器使之能运行PHP应用。

（4）配置PHP运行环境来调整性能并提高安全性。

（5）编辑wp-config.php文件来配置WordPress应用。

（6）编辑SMTP服务器的配置，确保只有虚拟服务器能发送邮件，以免被垃圾邮件发送者滥用。

（7）启动MySQL、SMTP和HTTP服务。

第1~2步处理安装及更新可执行程序。这些可执行程序在第3~6步被配置。第7步启动这些服务。

系统管理员经常根据操作指南手动进行这些步骤。不推荐手动部署应用在灵活的云环境中。相反，我们的目标是使用接下来介绍的各种工具来使这些步骤自动化。

## 5.1 在灵活的云环境中部署应用程序

如果想利用云的优势，例如，根据当前负载调节服务器的数目或是搭建一个高可用的架构，用户需要在一天内多次启动新服务器。除此之外，用户也需要更新数量不断增长的虚拟服务器。部署应用所需的步骤并不会改变，如图5-1所示，用户只需要在多个服务器上进行操作。随着发展，手动部署软件到不断增长的服务器将变得不大现实，并且有很高的人为失败的风险。这就是为什么我们推荐使应用部署自动化的原因。

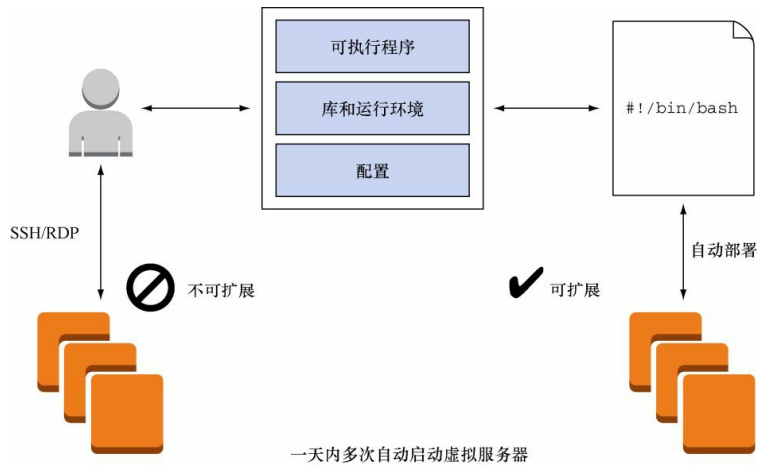


图5-1 在灵活的规模可变的云环境中部署必须是自动化的

在自动化部署流程中的投资将来会通过提高效率以及减少人为失败得到回报。



## 5.2 使用CloudFormation在服务器启动时运行脚本

在服务器启动的时候运行一个脚本是简单、有用并且灵活地进行自动化部署的方法。要把仅有操作系统的服务器完全安装并且配置好，需要遵循下列3个步骤。

- (1) 启动一台仅有操作系统的虚拟服务器。
- (2) 在引导程序完成后执行一个脚本。
- (3) 使用脚本安装并配置应用程序。

首先用户需要选择自己的虚拟服务器所使用的AMI。AMI为用户的虚拟服务器捆绑了操作系统以及预先安装好的软件。当用户从一个仅包含了操作系统，没有安装任何额外软件的AMI启动自己的虚拟服务器时，需要在引导程序结束时对虚拟服务器进行准备工作。把必要的安装和配置应用程序的步骤写成脚本能自动化这一任务。但是怎么在虚拟服务器引导结束后自动执行这个脚本呢？

### 5.2.1 在服务器启动时使用用户数据来运行脚本

在每一台虚拟服务器上用户可以插入一小段，不超过16 KB，被称为用户数据的数据。用户可以在创建一台新的虚拟服务器时指定用户数据。大多数AMI，如Amazon Linux Image和Ubuntu AMI都包含了这一典型的运行用户数据的功能。无论何时当用户基于这些AMI启动一台虚拟服务器时，用户数据在引导进程结束时被作为shell脚本被执行。执行的时候利用root用户的权限。

在虚拟服务器上，用户数据可以通过向一个特定URL进行HTTP Get 请求来获得。这个URL是<http://169.254.169.254/latest/user-data>，只能从这台虚拟服务器自己访问到。正如下面的例子中读者将看到的，我们能够通过作为脚本被执行的用户数据的帮助部署任何类型的应用程

序。

## 5.2.2 在虚拟服务器上部署OpenSwan作为VPN服务器

如果在咖啡店内使用笔记本电脑通过Wi-Fi工作，你可能希望让自己的网络流量通过VPN隧道在互联网上传输。这里将介绍如何使用用户数据与shell脚本在一台虚拟服务器上部署一台VPN服务器。我们使用的VPN解决方案叫作OpenSwan，它提供基于IPSec的通道，在Windows、OS X和Linux上都可以很容易地使用。图5-2展示了安装的示例。

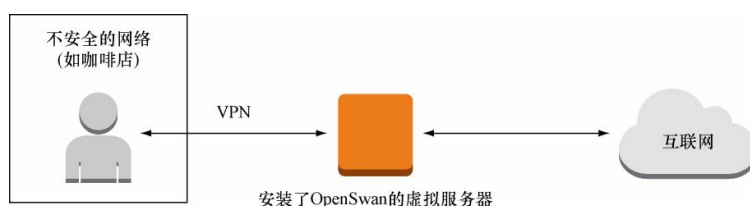


图5-2 在虚拟服务器上使用OpenSwan来传送个人计算机的数据流量

打开命令行，然后一步步执行代码清单5-1中的命令来启动一台虚拟服务器，并且在上面部署一个VPN服务器。我们已经准备好了一个CloudFormation模板来启动虚拟服务器以及它的依赖项。

代码清单5-1 在虚拟服务器上部署VPN服务器：CloudFormation与shell脚本

```
$ VpcId=$(aws ec2 describe-vpcs --query Vpcs[0].VpcId --output text)    ←
--获取默认VPC

$ SubnetId=$(aws ec2 describe-subnets --filters Name=vpc-id,Values=$VpcId \
    ←--获取默认子网
--query Subnets[0].SubnetId --output text)

$ SharedSecret=$(openssl rand -base64 30)    ←--创建一个随机密码（如果opens
sl不工作，创建你自己的随机序列）

$ Password=$(openssl rand -base64 30)    ←--创建一个随机共享密钥（如果opens
sl 不工作，创建你自己的随机序列）

$ aws cloudformation create-stack --stack-name vpn --template-url \    ←
--创建一个CloudFormation 堆栈
```

```
https://s3.amazonaws.com/awsinaction/chapter5/vpn-cloudformation.json \
--parameters ParameterKey=KeyName,ParameterValue=mykey \
ParameterKey=VPC,ParameterValue=$VpcId \
ParameterKey=Subnet,ParameterValue=$SubnetId \
ParameterKey=IPSecSharedSecret,ParameterValue=$SharedSecret \
ParameterKey=VPNUser,ParameterValue=vpn \
ParameterKey=VPNPassword,ParameterValue=$Password

$ aws cloudformation describe-stacks --stack-name vpn \
--query Stacks[0].Outputs      <--如果状态不是COMPLETE, 请在1 min 后重试
```

### OS X和Linux捷径

使用下列命令下载bash脚本并直接在本地机器执行下载脚本，可以避免手工在命令行中录入这些命令。该bash脚本包含的步骤与代码清单5-1所示相同：

```
$ curl -s https://raw.githubusercontent.com/AWSinAction/\
code/master/chapter5/\
vpn-create-cloudformation-stack.sh | bash -ex
```

最后一行命令的输出应该会列出公有VPN服务器的公有IP地址、共享密钥、VPN用户名和VPN密码。用户可以使用这一信息来从自己的计算机中建立VPN连接：

```
[...]
[
  {
    "Description": "Public IP address of the vpn server",
    "OutputKey": "ServerIP",
    "OutputValue": "52.4.68.225"
  },
  {
    "Description": "The shared key for the VPN connection (IPSec)",
    "OutputKey": "IPSecSharedSecret",
    "OutputValue": "sqmvJ1l/13bD6YqpmsKkPSMs9RrPL8itpr7m5V8g"
  },
  {
    "Description": "The username for the vpn connection",
    "OutputKey": "VPNUser",
    "OutputValue": "vpn"
  },
]
```

```
{
  "Description": "The password for the vpn connection",
  "OutputKey": "VPNPassword",
  "OutputValue": "aZQVFufF1UjJkesUfDmMj6DcHrWjuKShyFB/d01E"
}
]
```

让我们再仔细看一下VPN服务器的部署过程。我们将深入下面那些至今不经意间使用了的任务。

- 使用自定义用户数据启动一台虚拟服务器，并使用AWS CloudFormation为这台虚拟服务器配置防火墙。
- 在引导程序结束时执行一个shell脚本，通过程序包管理器来安装应用程序及其依赖项，并且编辑配置文件。

## 1. 使用**CloudFormation**来启动虚拟服务器并使用用户数据

可以使用CloudFormation来启动一台虚拟服务器并且配置一个防火墙。VPN服务器模板包括一个装入用户数据的shell脚本，如代码清单5-2所示。

### **Fn::Join和Fn::Base64**

这个CloudFormation模板包含两个新函数，即**Fn::Join** 和**Fn::Base64**。使用**Fn::Join**，能使用一个分隔符把多个值连接成一个值：

```
{"Fn::Join": ["delimiter", ["value1", "value2", "value3"]]}
```

函数**Fn::Base64** 把输入编码成Base64格式。用户会需要这个函数，因为用户数据必须被编码成Base64：

```
{"Fn::Base64": "value"}
```

代码清单5-2 CloudFormation模板的一部分，使用用户数据来初始化一台虚拟服务器

```

{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "Starts an virtual server (EC2) with OpenSwan [...]",
  "Parameters": {      ←--参数，使模板复用成为可能
    "KeyName": {
      "Description": "key for SSH access",
      "Type": "AWS::EC2::KeyPair::KeyName"
    },
    "VPC": {
      "Description": "Just select the one and only default VPC.",
      "Type": "AWS::EC2::VPC::Id"
    },
    "Subnet": {
      "Description": "Just select one of the available subnets.",
      "Type": "AWS::EC2::Subnet::Id"
    },
    "IPSecSharedSecret": {
      "Description": "The shared secret key for IPSec.",
      "Type": "String"
    },
    "VPNUser": {
      "Description": "The VPN user.",
      "Type": "String"
    },
    "VPNPassword": {
      "Description": "The VPN password.",
      "Type": "String"
    }
  },
  "Resources": {      ←--描述虚拟服务器
    "EC2Instance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "InstanceType": "t2.micro",
        "SecurityGroupIds": [{"Ref": "InstanceSecurityGroup"}],
        "KeyName": {"Ref": "KeyName"},
        "ImageId": "ami-1ecae776",
        "SubnetId": {"Ref": "Subnet"},
        "UserData":      ←--为虚拟服务器定义一个shell 脚本作为用户数据
          {"Fn::Base64": {"Fn::Join": ["", [      ←--连接并对字符串进行编码
            "#!/bin/bash -ex\n",
            "export IPSEC_PSK=", {"Ref": "IPSecSharedSecret"}, "\n",
            "export VPN_USER=", {"Ref": "VPNUser"}, "\n",      ←--导出参数至
            "export VPN_PASSWORD=", {"Ref": "VPNPassword"}, "\n",
            "export STACK_NAME=", {"Ref": "AWS::StackName"}, "\n",
            "export REGION=", {"Ref": "AWS::Region"}, "\n",
            环境变量使它们能被接下来调用的外部shell脚本使用
          ]]}
        }
      }
    }
  }
}

```

```

        "curl -s https://.../vpn-setup.sh | bash -ex\n"      <--通过http 获
取shell 脚本并执行
    ]]]}
    },
    [...]
    },
    [...]
    },
    "Outputs": {
        [...]
    }
}

```

基本上，用户数据包含一个用来获取并执行真正的脚本的小脚本 `vpn-setup.sh`，真正的脚本包含所有的安装可执行程序以及配置服务的命令。这样做可以避免以可读性较差的格式插入JSON CloudFormation模板所需的脚本。

## 2. 使用脚本安装并配置一个VPN服务器

代码清单5-3中所示的 `vpn-setup.sh` 脚本通过程序包管理器 `yum` 安装程序包并且写一些配置文件。读者不必要理解VPN服务器配置的详细信息，只需要了解这个shell脚本在引导过程中被执行，它会安装并配置一台VPN服务器。

代码清单5-3 在服务器启动时安装程序包并写配置文件

```

#!/bin/bash -ex

[...]

PRIVATE_IP='url -s http://169.254.169.254/latest/meta-data/local-ipv4'
<--获取虚拟服务器私有IP 地址

PUBLIC_IP='curl -s http://169.254.169.254/latest/meta-data/public-ipv4'
<--获取虚拟服务器公有IP 地址

yum-config-manager --enable epel && yum clean all      <--向包管理器yum添加
额外程序包

yum install -y openswan xl2tpd      <--安装软件程序包

```

```
cat > /etc/ipsec.conf <<EOF      <--为IPSec 写一个包含共享密钥的文件
[...]
EOF

cat > /etc/ipsec.secrets <<EOF    <--为IPSec (OpenSwan) 写配置文件
$PUBLIC_IP %any : PSK "${IPSEC_PSK}"
EOF

cat > /etc/xl2tpd/xl2tpd.conf <<EOF    <--为L2TP 管道写配置文件
[...]
EOF

cat > /etc/ppp/options.xl2tpd <<EOF    <--为PPP 服务写配置文件
[...]
EOF

service ipsec start && service xl2tpd start      <--启动VPN 服务器需要的服务

chkconfig ipsec on && chkconfig xl2tpd on      <--配置VPN 服务器的运行等级
```

我们已经学习了如何使用EC2用户数据与一个shell脚本在一台虚拟服务器上部署一个VPN服务器。在终止虚拟服务器之后，我们将准备学习如何部署一个普通网站应用，而不需要自定义脚本。

#### 资源清理

我们已经完成了VPN服务器的示例，别忘了终止虚拟服务器并且清理环境。需要做的是：在终端输入`aws cloudformation delete-stack --stack-name vpn`。

## 5.2.3 从零开始，而不是更新已有的服务器

在本节中，我们学习了如何使用用户数据来部署一个应用。用户数据中的脚本在引导过程结束时被执行。但怎么用这个方法更新应用呢？

我们实现了在自己的虚拟服务器引导流程时自动化安装与配置软件，所以可以启动一个新的虚拟服务器而不需要增加额外的工作。如果

必须更新自己的应用或它的依赖项，可以按以下步骤来做。

（1）确保应用或软件的最新的版本可以通过操作系统的程序包库获得，或者编辑用户数据脚本。

（2）基于CloudFormation模板及用户数据脚本启动一台新的虚拟服务器。

（3）测试部署到新的虚拟服务器上的应用。如果一切正常，则继续操作下一步。

（4）切换负载到新的虚拟服务器（如通过更新DNS记录）。

（5）终止旧的虚拟服务器，且扔掉它不用的依赖项。



## 5.3 使用Elastic Beanstalk部署一个简单的网站应用

如果必须部署一个普通网站应用，不需要从头开始。AWS提供了一项服务可以帮助用户部署基于PHP、Java、.NET、Ruby、Node.js、Python、Go和Docker的网站应用，它被称作AWS Elastic Beanstalk。使用Elastic Beanstalk，用户就不必操心自己的操作系统或虚拟服务器，因为它在它们之上加了一个抽象层。

Elastic Beanstalk帮助用户处理下面反复发生的任务。

- 为网站应用（PHP、Java等）提供一个运行环境。
- 自动安装并更新网站应用。
- 配置网站应用及其环境。
- 调整网站应用规模来负载均衡。
- 监控和调试网站应用。

### 5.3.1 Elastic Beanstalk的组成部分

了解Elastic Beanstalk的不同组成部分有助于了解它的功能。图5-3展示了这些元素。

- 应用 是一个逻辑上的容器。它包含了版本、环境和配置。如果用户在一个区域开始使用Elastic Beanstalk，首先需要创建一个应用。
- 版本 包含用户的应用的指定版本。要创建一个新版本，用户必须上传自己的可执行文件（打包成一个压缩文档）到用来存储静态文件的Amazon S3服务。版本是一个指向这个可执行文件的压缩文档的指针。
- 配置模板 包含默认配置。用户可以通过自定义的配置模板管理自己的应用的配置（如应用监听的端口）以及环境配置（如虚拟服务器的大小）。
- 环境 是Elastic Beanstalk执行应用的地方。它由版本和配置构成。用户可以通过多次使用版本和配置为一个应用运行多个环境。

目前理论已经足够。让我们继续来部署一个简单的网站应用。

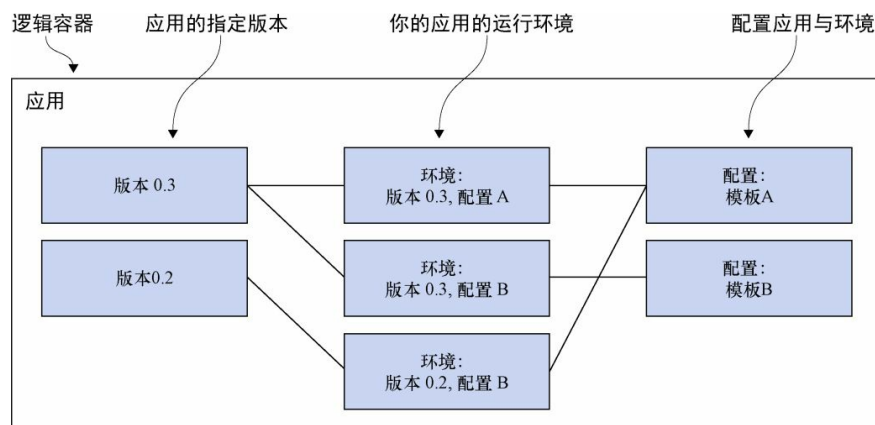


图5-3 Elastic Beanstalk应用包含了版本、配置和环境

## 5.3.2 使用Elastic Beanstalk部署一个Node.js应用Etherpad

使用错误的工具来协作编辑文档可能很痛苦。Etherpad是一个开源的在线编辑器，它让许多人可以同时编辑一份文档。我们将通过以下3个步骤在Elastic Beanstalk的帮助下部署这个基于Node.js的应用。

- (1) 创建应用：逻辑上的容器。
- (2) 创建版本：指向特定Etherpad版本的指针。
- (3) 创建环境：Etherpad运行的地方。

### 1. 为AWS Elastic Beanstalk创建应用

打开命令行并且执行下面的命令来为Elastic Beanstalk服务创建一个应用：

```
$ aws elasticbeanstalk create-application --application-name etherpad
```

在AWS Elastic Beanstalk的帮助下，我们已经为所有其他部署Etherpad所必需的组件创建了一个容器。

## 2. 为AWS Elastic Beanstalk创建版本

我们可以使用下面的命令创建自己的Etherpad应用的新版本：

```
$ aws elasticbeanstalk create-application-version \  
--application-name etherpad --version-label 1.5.2 \  
--source-bundle S3Bucket=awsinaction,S3Key=chapter5/etherpad.zip
```

针对这个示例，我们上传了一个包含了Etherpad版本1.5.2的zip压缩文档。如果读者想部署另一个应用，可以上传自己的应用的静态文件至AWS S3服务。

## 3. 用Elastic Beanstalk创建一个环境来执行Etherpad

要使用Elastic Beanstalk部署Etherpad，需要基于Amazon Linux及Etherpad的版本为Node.js创建一个环境。要获取最新的Node.js环境版本，列出包含它的解决方案堆栈名（solution stack name），运行下面的命令：

```
$ aws elasticbeanstalk list-available-solution-stacks --output text \  
--query "SolutionStacks[?contains(@, 'running Node.js')] | [0]"\  
  
64bit Amazon Linux 2015.03 v1.4.6 running Node.js
```

选项**EnvironmentType = SingleInstance** 自动启动一个不可变规模且无负载均衡的单台虚拟服务器。使用从前一个命令得到的输出替换**\$SolutionStackName**：

```
$ aws elasticbeanstalk create-environment --environment-name etherpad \  
--application-name etherpad \  
--option-settings Namespace=aws:elasticbeanstalk:environment,\
```

```
OptionName=EnvironmentType,Value=SingleInstance \
--solution-stack-name "$SolutionStackName" \
--version-label 1.5.2
```

## 4. 玩转Etherpad

我们已经为Etherpad创建了一个环境。在将浏览器指向我们的Etherpad安装前，需要花费几分钟用下面的命令来跟踪Etherpad环境的状态：

```
$ aws elasticbeanstalk describe-environments --environment-names etherpad
```

如果Status变为Ready，并且Health变成Green，说明已经准备好了，可以创建我们的第一个Etherpad文档了。命令describe 的输出应该与代码清单5-4所示的例子类似。

代码清单5-4 描述Elastic Beanstalk的状态

```
{
  "Environments": [{
    "ApplicationName": "etherpad",
    "EnvironmentName": "etherpad",
    "VersionLabel": "1",
    "Status": "Ready",          ←--等待Status 变为Ready
    "EnvironmentId": "e-pwbfmgrsjp",
    "EndpointURL": "23.23.223.115",
    "SolutionStackName": "64bit Amazon Linux 2015.03 v1.4.6 running Node.js",
    "CNAME": "etherpad-cxzshvfjzu.elasticbeanstalk.com",    ←--环境的DNS
    "Health": "Green",          ←--等待Health变为Ready
    "Tier": {
      "Version": " ",
      "Type": "Standard",
      "Name": "WebServer"
    },
    "DateUpdated": "2015-04-07T08:45:07.658Z",
    "DateCreated": "2015-04-07T08:40:21.698Z"
  ]
}
```

```
}]  
}
```

我们已经利用3个命令在AWS上部署了一个Node.js网站应用。现在把浏览器指向CNAME中的URL，并输入一个新文档名，点击OK按钮来打开一个新文档。图5-4展示了正在使用的Etherpad文档。

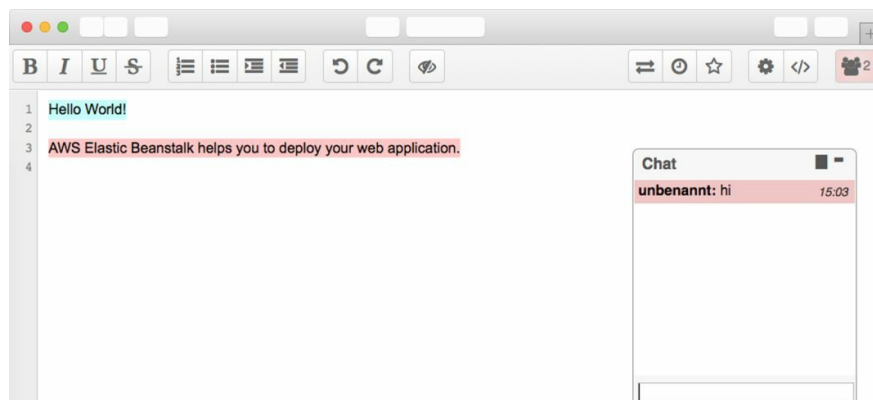


图5-4 使用中的在线文本编辑器Etherpad

## 5. 使用管理控制台探索Elastic Beanstalk

我们已经使用Elastic Beanstalk和AWS CLI创建了应用、版本和环境，部署了Etherpad。用户也可以通过管理控制台（一个基于网页的用户界面）来控制Elastic Beanstalk服务。

- （1）打开AWS管理控制台。
- （2）在导航栏中点击“服务”，然后点击Elastic Beanstalk服务。
- （3）点击Etherpad环境，以一个绿色框表示。将显示Etherpad应用的概况，如图5-5所示。

我们也可以通过Elastic Beanstalk从自己的应用中获取日志信息。使用下面的步骤下载最新的日志信息。

- （1）在子菜单中选择“日志”，我们会看见一个图5-6所示的界面。

(2) 点击“请求日志”，然后选择最后100行。

(3) 几秒钟后，表中将显示一个新的入口。点击“下载”将日志文件下载到我们的计算机上。

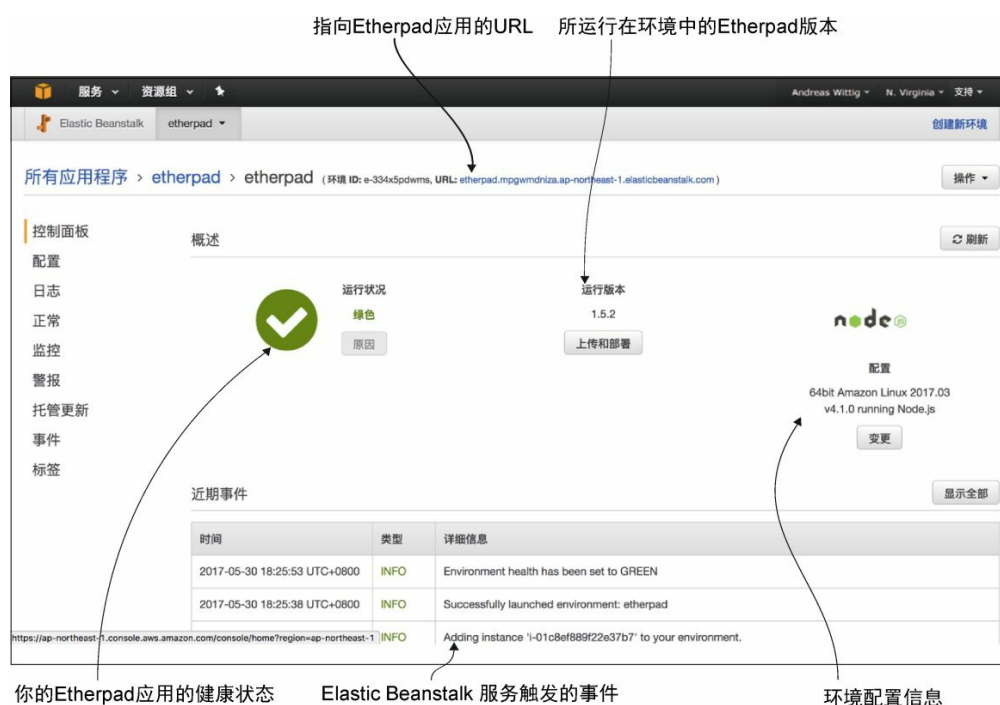


图5-5 运行Etherpad的AWS Elastic Beanstalk环境总览

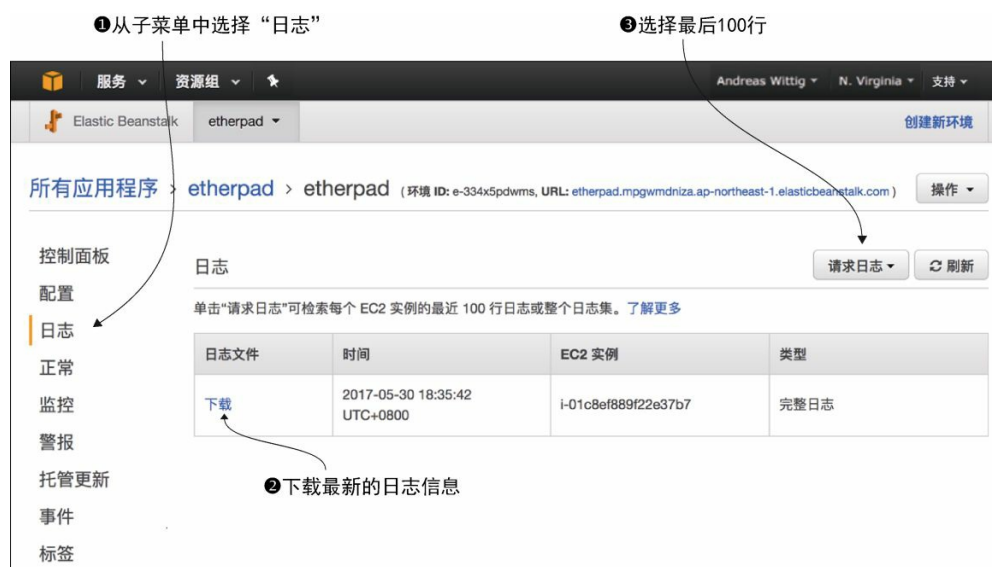


图5-6 通过AWS Elastic Beanstalk从Node.js应用中下载日志信息

## 资源清理

现在你已经成功地在AWS Elastic Beanstalk的帮助下部署了Etherpad，并且学习了这个服务的不同组件，现在是时候清理了。运行下面的命令来终止Etherpad环境：

```
$ aws elasticbeanstalk terminate-environment --environment-name etherpad
```

可以使用下面的命令检查环境的状态：

```
$ aws elasticbeanstalk describe-environments --environment-names etherpad
```

一直等到状态变为Terminated，然后继续执行下面的命令：

```
$ aws elasticbeanstalk delete-application --application-name etherpad
```

就是这样。我们已经终止了提供环境给Etherpad的虚拟服务器，并且删除了Elastic Beanstalk中的所有相关组件。

## 5.4 使用OpsWorks部署多层架构应用

使用Elastic Beanstalk部署一个基本网站应用很方便。但是，如果用户需要部署一个更复杂的包含不同服务的应用（也称多层应用），将会受到Elastic Beanstalk的限制。在本节中，我们将学习AWS OpsWorks，一个有AWS提供的免费的可以帮助用户部署多层架构应用的服务。

OpsWorks帮助用户控制AWS资源如虚拟服务器、负载均衡器和数据库，并且让用户可以部署应用。这一服务提供在下面环境中的标准层：

- HAProxy（负载均衡器）；
- PHP（应用服务器）；
- MySQL（数据库）；
- Rails app server（Ruby on Rails）；
- Java app server（Tomcat服务器）；
- Memcached（内存缓存）；
- 静态Web服务器；
- AWS Flow（Ruby）；
- Ganglia（监控）。

用户也可以添加一个自定义层来部署想要的任何内容。部署是在Chef的帮助下进行控制的，Chef是一个配置管理工具。Chef使用在cookbook中组织的recipe来部署应用到任意系统中。用户可以使用标准recipe或自行创建。

### 关于Chef

Chef是一个类似于Puppet、SaltStack和Ansible的配置管理工具。Chef将用领域特定语言（domain-specific language, DSL）写的模板（recipe）转换成动作，来配置及部署应用。recipe可以包含用于安装的程序包、可运行的服务或者可写的配置文件。相关的recipe可以存放到cookbook中集中管理。Chef分析现状并在必要时更改资源，以达到recipe中描述的状态。

读者可以在Chef的帮助下复用他人的cookbook和recipe。社区中发布了各种开源代码许可下的cookbook和recipe。

Chef可以单独运行或使用客户端/服务器模式。在客户端/服务器模式下，它作为一个集群管理工具，可以帮助用户管理一个由很多虚拟服务器构成的分布式系统。在单机模式下，



用户可以在单个虚拟服务器上执行recipe。AWS OpsWorks使用单机运行模式时，集成了自己的集群管理组件，不需要用户在客户端/服务器模式中的烦琐配置与安装。

除了帮助用户部署应用，OpsWorks还有助于用户更好地扩展、监控和更新运行在不同逻辑层下的虚拟服务器。

## 5.4.1 OpsWorks的组成部分

了解OpsWorks的不同组件有助于了解它的功能。图5-7展示了这些元素。

- 堆栈 是一个所有其他OpsWorks组件的容器。用户能够创建一个或多个堆栈，并且为每个堆栈添加一个或多个层，可以使用不同的堆栈来区分（如产品环境与测试环境），也可以使用不同的堆栈来区分不同的应用。
- 层 属于堆栈。一个层代表一个应用，也可以称为服务。OpsWorks为标准的网站应用，如PHP和Java，提供预定义好的层，但是用户可以为任何自己能想到的应用自由运用自定义堆栈。一个层负责配置预发布软件到虚拟服务器上。用户可以向一个层添加一台或多台虚拟服务器。在这里虚拟服务器被称作实例。
- 实例 代表了虚拟服务器。用户可以在每一层启动一个或多个实例。用户可以使用不同版本的Amazon Linux和Ubuntu，或一个自定义的AMI作为实例的基础，然后为规模伸缩定义基于负载或时间的规则来启动与终止实例。
- 应用程序 是你要部署的软件。OpsWorks自动将用户的应用程序部署到一个合适的层。用户可以从Git或Subversion库，或通过HTTP作为压缩存档获取应用程序。OpsWorks帮助用户将自己的应用程序安装和更新到一个或多个实例上。

让我们来看看怎样在OpsWorks的帮助下部署一个多层架构的应用。

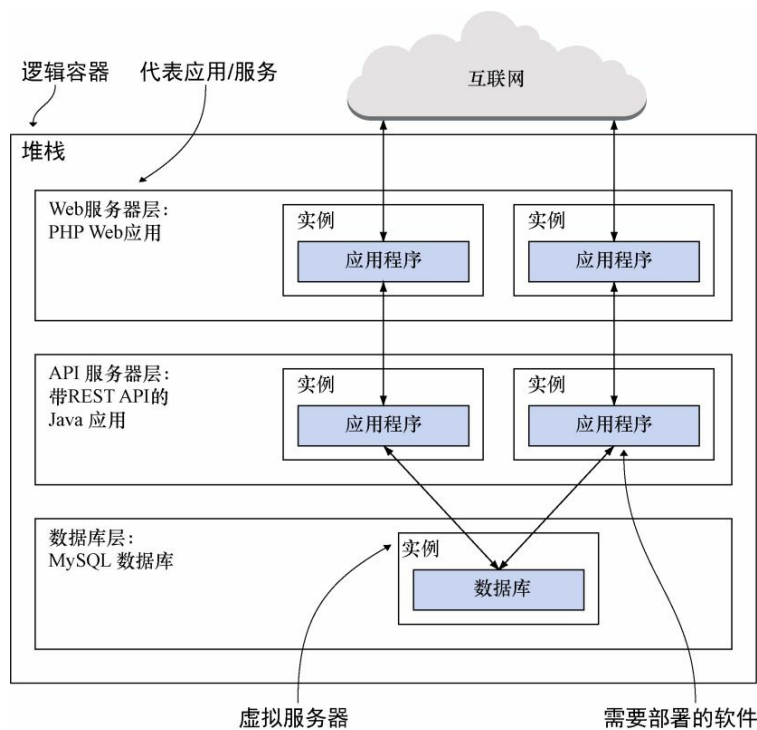


图5-7 堆栈、层、实例和应用程序是OpsWorks的主要组件

## 5.4.2 使用OpsWorks部署一个IRC聊天应用

IRC（Internet Relay Chat）依然是流行的通信手段。在本节中，我们将部署一个基于网站的IRC客户端kiwiIRC和我们自己的IRC服务器。图5-8展示了如何搭建一个分布式系统，包含了一个网站应用并提供IRC客户端和IRC服务器。

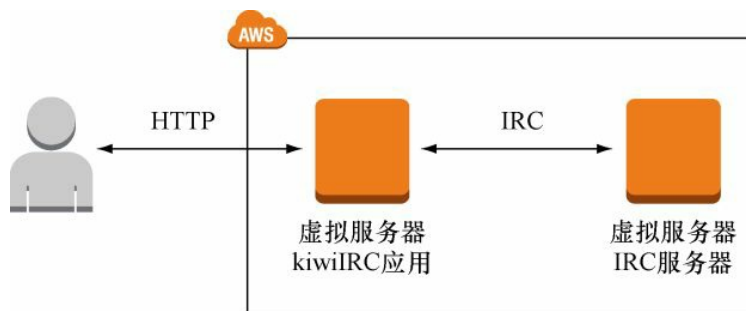


图5-8 搭建由一个网站应用与一台IRC服务器构成的IRC架构

kiwiIRC是用JavaScript为Node.js编写的一个开源网站应用。下面是

在OpsWorks的帮助下部署一个两层架构应用所必需的步骤。

- (1) 创建一个堆栈，所有其他组件的容器。
- (2) 为kiwiIRC创建一个Node.js层。
- (3) 为IRC服务器创建一个自定义层。
- (4) 创建一个应用程序，将kiwiIRC部署到Node.js层。
- (5) 为每一层添加一个实例。

接下来我们将学习如何在管理控制台上完成这些步骤。我们也可以通过命令行控制OpsWorks，就像使用Elastic Beanstalk或CloudFormation时所做的那样。

## 1. 创建一个新的**OpsWorks**堆栈

打开管理控制台，然后创建一个新的堆栈。图5-9展示了必要的步骤。

选择默认VPC，这个列表中的唯一选项      IRC 服务器程序包，默认在Ubuntu服务器上可用

### Create a stack with instances that run Linux and Chef 11.10

Classic experience. Use our built-in cookbooks for layers, applications & deployments to get started. Use your own Chef cookbooks to override or extend the built-in layers. [Learn more.](#)

**Stack name**

**Region**

**VPC**

**Default subnet**

**Default operating system**  *Need a different OS? [Let us know.](#)*

**Default SSH key**

**Chef version**

**Use custom Chef cookbooks** ☐ No *Define the source of your Chef cookbooks*

**Stack color**

**Advanced options**

**Default root device type** ☒ EBS backed ☐ Instance store

**IAM role**

**Default IAM instance profile**

**API endpoint region** NEW

**Hostname theme**

**OpsWorks Agent version**

通过SSH连接调试你的服务器需要一个SSH密钥

使用层名字为你的应用服务器命名，如Node.js应用程序

图5-9 在OpsWorks上创建一个新的堆栈

- (1) 点击Select Stack下的Add Stack或Add Your First Stack。
- (2) 在Name中输入irc。
- (3) 在Region中选择US East (N. Virginia)。
- (4) 默认VPC是唯一可用的，选中它。
- (5) Default Subnet，选择us-east-1a。
- (6) Default Operating System，选择Ubuntu 14.04 LTS。
- (7) Default Root Device Type，选择EBS Backed。
- (8) IAM Role，选择New IAM Role。这样做会自动创建所需的依赖。

(9) 选择用户的SSH密钥mykey，作为Default SSH Key。

(10) 展开高级设置。Default IAM Instance Profile，选择New IAM Instance Profile。这样做会自动创建所需的依赖。

(11) Hostname Theme，选择Layer Dependent。虚拟服务器会依据它们所在的层来命名。

(12) 点击Add Stack来创建堆栈。

用户被重定向到自己的irc堆栈的概述。每件事情都准备好了，就可以创建第一个层了。

## 2. 为OpsWorks堆栈创建一个Node.js层

kiwiIRC是一个Node.js应用，因而我们需要为irc堆栈创建一个Node.js层。按照以下步骤来操作。

(1) 从子菜单中选择Layers。

(2) 点击Add Layer按钮。

(3) 对Layer Type选择Node.js App Server，如图5-10所示。

(4) 选择Node.js最新版本0.12.x。

(5) 点击Add layer。

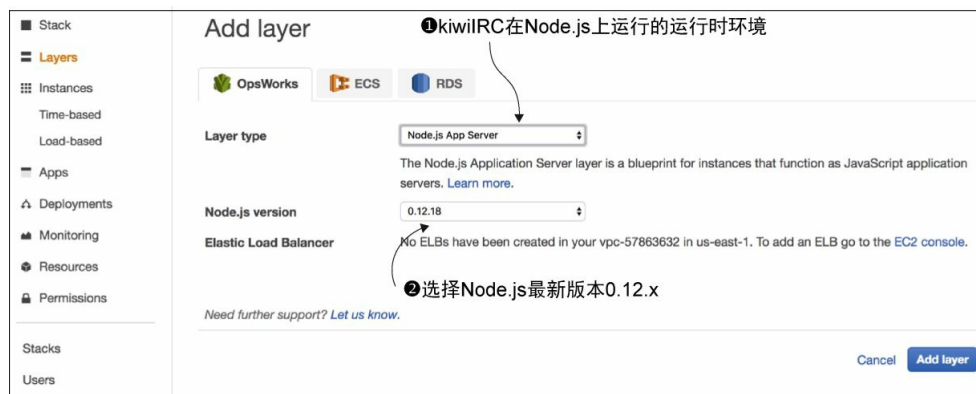


图5-10 为kiwiIRC创建一个Node.js层

这里已经创建了一个Node.js层。现在我们需要重复这些步骤来添加另一个层，并部署自己的IRC服务器。

### 3. 为OpsWorks堆栈创建一个自定义层

一个IRC服务器不是一个典型的网站应用，因此不可能使用默认层类型。我们将用一个自定义层来部署IRC服务器。Ubuntu程序包库包含了各种IRC服务器实现，这里将使用ircd-ircu 程序包。按照下列步骤为IRC服务器创建一个自定义堆栈。

- (1) 从子菜单中选择Layers。
- (2) 点击Add Layer。
- (3) 对Layer Type选择Custom，如图5-11所示。
- (4) 对Name和Short Name中输入irc-server 。
- (5) 点击Add Layer。



图5-11 创建一个自定义层来部署IRC服务器

这里已经创建了一个自定义层。

IRC服务器需要通过端口6667访问。要允许访问这一端口，需要定义一个自定义防火墙。执行代码清单5-5中的命令来为自己的IRC服务器

创建一个自定义防火墙。

代码清单5-5 使用CloudFormation创建一个自定义防火墙

```
$ aws ec2 describe-vpcs --query Vpcs[0].VpcId --output text      <--- 获取默认
VPC, 记作$VpcId

$ aws cloudformation create-stack --stack-name irc \            <--- 创建一个CloudFo
rmation堆栈
--template-url https://s3.amazonaws.com/awsinaction/\
chapter5/irc-cloudformation.json \
--parameters ParameterKey=VPC,ParameterValue=$VpcId

$ aws cloudformation describe-stacks --stack-name irc \

--query Stacks[0].StackStatus      <--- 如果状态不是COMPLETE, 10 s 后再执行一遍
这条命令
```

#### OS X和Linux的快捷方式

使用下面的命令行下载一个bash脚本并直接在本地机器上执行, 这样能避免在命令行中手动输入这些命令。这个bash脚本包含了代码清单5-5所示的相同步骤:

```
$ curl -s https://raw.githubusercontent.com/AWSinAction/\
code/master/chapter5/irc-create-cloudformation-stack.sh \
| bash -ex
```

接下来要将这个自定义防火墙配置关联到自定义OpsWorks层。按照下面的步骤操作。

- (1) 从子菜单中选择Layers。
- (2) 点击并打开irc-server layer。
- (3) 切换至Security标签并点击Edit。

(4) 在Custom Security Groups中选择开头为irc的安全组, 如图5-12所示。

(5) 点击Save。



图5-12 向IRC服务器层添加一个自定义防火墙配置

我们需要为IRC服务器配置的最后一件事是：用层recipes部署IRC服务器。按照下面的步骤操作。

- (1) 从子菜单中选择Layers。
- (2) 点击并打开irc-server layer。
- (3) 切换至Recipes标签并点击Edit。
- (4) 在OS Packages中添加程序包ircd-ircu，如图5-13所示。
- (5) 点击+按钮，然后点击Save按钮。

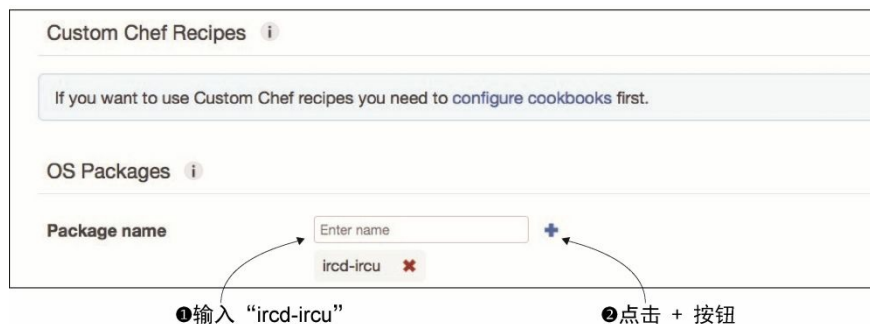


图5-13 向自定义层添加一个IRC程序包



我们已经成功地创建并配置了一个自定义层来部署IRC服务器。接下来我们将在OpsWorks中将kiwiIRC网站应用添加为一个应用程序。

#### 4. 向Node.js层添加一个应用程序

OpsWorks可以向默认层部署应用程序。我们已经创建了一个Node.js层。按照下面的步骤向这个层添加一个应用程序。

- (1) 从子菜单中选择Apps。
- (2) 点击Add an App按钮。
- (3) 在Name中输入kiwiIRC。
- (4) 在Type中选择Node.js
- (5) 为Repository Type选Git，并为Repository URL输入<https://github.com/AWSinAction/KiwiIRC.git>，如图5-14所示。
- (6) 点击Add App按钮。

我们的第一个OpsWorks堆栈现在完全配置好了。还漏掉了一件事——启动一些实例。

①为应用程序选择一个名字

②选择Node.js 作为环境

③访问公开的GitHub库

### Add App

Settings

Name: kiwiIRC

Type: Node.js

By default we expect your Node.js app to listen on port 80. Furthermore, the file we pass to node has to be named "server.js" and should be located in your app's root directory.

Data Sources

Data source type: ☐ RDS ☐ OpsWorks ☒ None

Application Source

Repository type: Git

Repository URL: https://github.com/AWSInAction/KiwiIRC Set the URL where your repository can be accessed.

Repository SSH key: Optional Enter the SSH key required to access a private repository.

Branch/Revision: Optional

图5-14 向OpsWorks添加kiwiIRC，一个Node.js应用程序

## 5. 添加实例来运行IRC客户端与服务

添加两个实例来实现kiwiIRC客户端与服务。向一个层添加新实例很容易，按照下面的步骤来操作。

- (1) 从子菜单中选择Instances。
- (2) 在Node.js App Server层中点击Add an Instance按钮。
- (3) 在Size中选择t2.micro，最小、最便宜的虚拟服务器，如图5-15所示。
- (4) 点击Add Instance。

我们已经向Node.js应用程序服务器层添加了一个实例。为irc-server层重复这些步骤。

实例的概览应该类似于图5-16。要启动这些实例，点击Start。

虚拟服务器引导并部署需要花费一些时间。

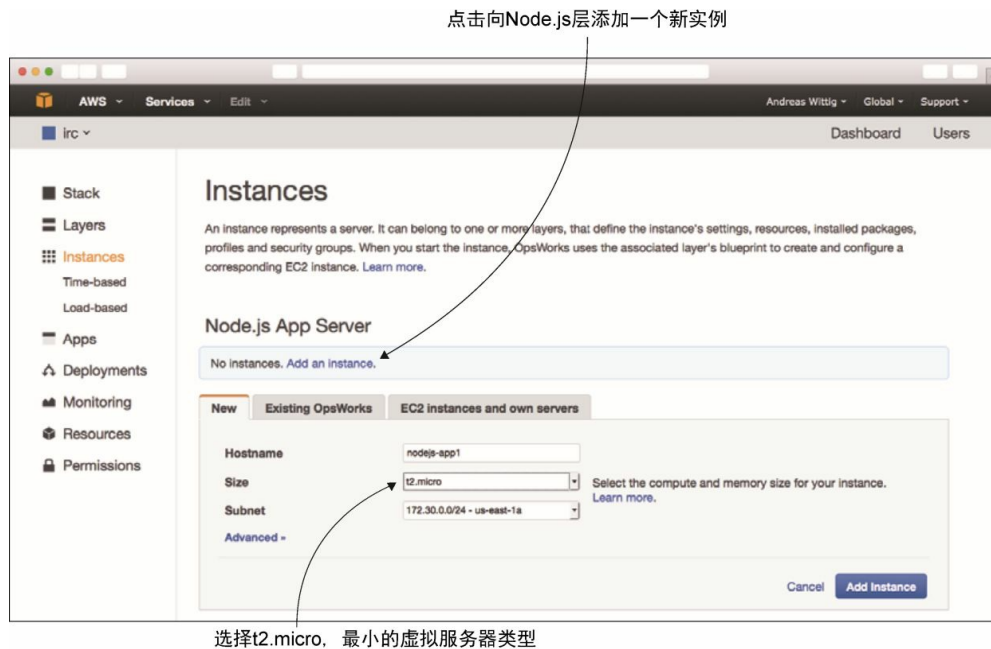


图5-15 向Node.js层添加一个新实例

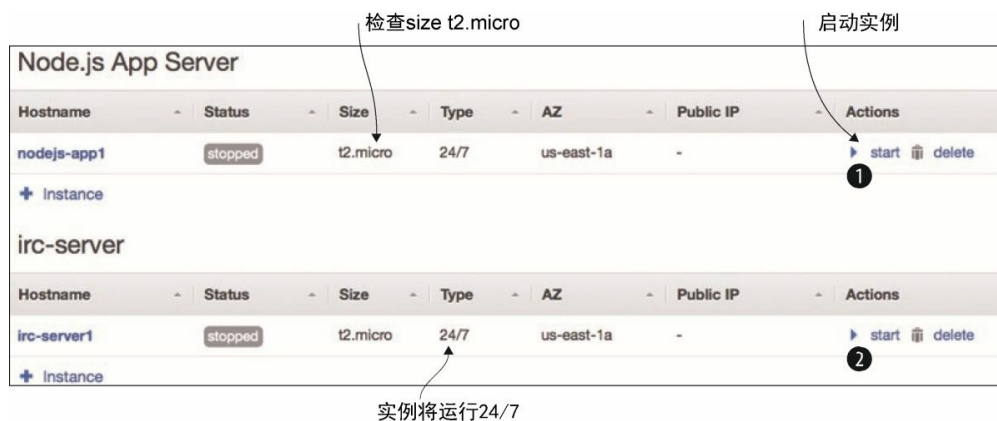


图5-16 启动IRC网站客户端与服务器实例

## 6. 玩转kiwiIRC

启动两台实例后，耐心等待两个实例的状态都变为Online，如图5-17所示。然后就可以按下面的步骤在浏览器中打开kiwiIRC了。

(1) 记住实例irc-server1的公有IP地址。稍后我们需要用它来连接IRC服务器。

(2) 点击nodejs-app1实例的公有IP地址来在浏览器的新标签中打开kiwiIRC网站应用。



图5-17 等待部署完成并在浏览器中打开kiwiIRC

kiwiIRC应用应该会在浏览器中装载，然后我们应该看见一个图5-18所示的登录界面。按照下面的步骤使用kiwiIRC网站客户端登录到IRC服务器：

- (1) 输入一个昵称。
- (2) 在Channel项中输入#awsinaction。
- (3) 点击Server and Network，打开连接详细信息。
- (4) 在Server项中输入irc-server1的IP地址。
- (5) 在Port项中输入6667。
- (6) 禁用SSL。
- (7) 点击Start，然后等待几秒。

目前为止，我们已经在AWS OpsWorks的帮助下部署了一个基于网站的IRC客户端和服务端。



图5-18 使用kiwiIRC登录到IRC服务器，使用信道#awsinaction

#### 资源清理

是时候做一些清理工作了。按照下面的步骤避免意外费用支出。

- (1) 使用管理控制台打开OpsWorks服务。
- (2) 点击并选择irc堆栈。
- (3) 从子菜单中选择Instances。
- (4) 停止两个实例，并等待直到它们的状态变为Stopped。
- (5) 删除两个实例，并等待知道它们从概览中消失。
- (6) 从子菜单中选择Apps。
- (7) 删除kiwiIRC应用程序。
- (8) 从子菜单中选择Stack。
- (9) 点击Delete Stack按钮，并确认删除。
- (10) 从终端执行`aws cloudformation delete-stack --stack-name irc`。

## 5.5 比较部署工具

在本章中我们使用了以下3种方法来部署应用。

- 使用AWS CloudFormation在服务器启动时运行一个脚本。
- 使用AWS Elastic Beanstalk部署一个通用网站应用。
- 使用AWS OpsWorks部署一个多层架构应用。

本节中我们将讨论这几种方法的不同之处。

### 5.5.1 对部署工具分类

图5-19分类了3个AWS部署选项。在AWS Elastic Beanstalk帮助下部署一个应用所需的工作量较低。要从这一点获益，用户的应用必须符合Elastic Beanstalk的惯例。例如，应用必须运行在某一标准运行环境中。如果用户使用OpsWorks，就能够有更多自由来根据自己的应用需求调整服务。例如，用户可以部署互相依赖的不同层，也可以使用自定义层来在Chef recipe的帮助下部署任何应用，这需要额外的工作量，但让用户获得了更多的自由度。此外，用户可以使用CloudFormation，通过在引导流程结束时运行一个脚本来部署应用。在CloudFormation的帮助下用户可以部署任何应用。这一方法的缺点是，因为用户没有使用标准工具，所以用户需要做更多的工作。

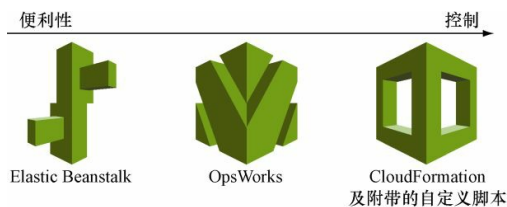


图5-19 比较AWS上不同的部署应用的方式

### 5.5.2 比较部署服务

上面的分类能帮助用户确定部署应用的最适合方法。表5-1中的比较强调了其他重要的考量因素。

表5-1 使用**CloudFormation**在服务器启动时运行脚本、**Elastic Beanstalk**和**OpsWorks**的区别

	使用 <b>CloudFormation</b> 在服务器启动	<b>Elastic Beanstalk</b>	<b>OpsWorks</b>
配置管理工具	所有可用工具	专属	Chef
支持的平台	任意	<ul style="list-style-type: none"> <li>■ PHP</li> <li>■ Node.js</li> <li>■ IIS</li> <li>■ Java/Tomcat</li> <li>■ Python</li> </ul>	<ul style="list-style-type: none"> <li>■ Ruby on Rails</li> <li>■ Node.js</li> <li>■ PHP</li> <li>■ Java/Tomcat</li> <li>■ Custom/any</li> </ul>
支持的平台	任意	<ul style="list-style-type: none"> <li>■ Ruby</li> <li>■ Docker</li> </ul>	
支持的部署构件	任意	Amazon S3上的Zip存档文件	Git、SVN、存档文件（如Zip）
常见的使用场景	复杂且不标准的环境	普通网站应用	微服务环境
没有停机时间的更新	可能	是	是
供应商锁定效应	中等	高	中等

在AWS上还有许多其他部署应用的可选项，从开源软件到第三方服务。我们的建议是使用AWS部署服务之一，因为它们更好地集成了许多其他AWS服务。我们推荐使用CloudFormation的用户数据来部署应用，因为它是一种灵活的方法。在CloudFormation的帮助下管理Elastic Beanstalk和OpsWorks也是可能的。

一个自动化的部署流程将有助于更快地迭代与创新。用户会更经常地部署自己的应用的新版本。要避免服务中断，用户需要考虑自动化的测试软件与架构的变更，并且要能在必要时快速回滚到上一版本。



## 5.6 小结

- 因为在一个动态云环境中虚拟服务器变化更频繁，所以不建议手动部署应用到虚拟服务器上。
- AWS提供不同的工具帮助用户将应用部署到虚拟服务器上。使用这些工具可以避免从头开始。
- 如果能够进行自动化部署，可以无须照顾具体的服务器而直接更新一个应用。
- 在虚拟服务器启动时插入Bash或PowerShell脚本，可以让用户有区别地初始化服务器，例如，安装不同的软件或配置服务。
- OpsWorks在Chef的帮助下是部署多层架构应用的好方法。
- Elastic Beanstalk适合部署普通网站应用。
- CloudFormation在用户部署自己负责的应用时给其最多的控制能力。

## 第6章 保护系统安全：IAM、安全组和VPC

### 本章主要内容

- 安装软件更新
- 使用用户与角色来控制对AWS账户的访问
- 使用安全组来控制数据传输安全
- 使用CloudFormation来创建私有网络
- 谁该对安全负责？

假如安全是一堵墙，那么我们需要很多砖块来建造这堵墙。本章将着重讲述在AWS上保护用户系统安全所需的4个重要的砖块。

- 安装软件更新 ——每天都有新的软件安全漏洞被发现。软件制造商发布更新来修补这些漏洞，而用户的任务就是在这些更新发布后及时安装它们，否则用户的系统就容易成为黑客攻击的受害者。
- 限制访问AWS账户 ——假如用户不是唯一访问自己的AWS账户的人，那这一点就变得更重要了（如果同事以及脚本也会访问它的话）。一个有bug的脚本可以很容易就终止用户的所有EC2实例而不是那个用户想要终止的。授予最小权限是保护用户的AWS资源免于意外或故意的致命操作。
- 控制进出用户的EC2实例的网络传输 ——用户只想要那些必需的端口能够被访问到。如果用户运行一个网络服务器，对外部世界只需要为HTTP流量打开80端口，为HTTPS流量打开443端口。请关闭所有其他端口！
- 在AWS内创建一个私有网络 ——用户可以创建从互联网不可达的子网。如果这些子网不可达，就没人能访问它们。没有人能访问？好吧，我们将学习如何使自己能访问它们却能阻止别人这样做。

示例都包含在免费套餐中

本章中的所有示例都包含在免费套餐中。只要不是运行这些示例好几天，就不需要支付

任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

落了一块重要的砖块：保护用户自己开发的 ứng dụng 的安全。用户需要检查使用者的输入并且只允许必需的字符，不要使用明文存储密码，使用SSL来加密自己的服务器与使用者之间的数据传输，等等。

#### 本章的要求

要完全理解本章，读者应该熟悉以下概念：

- 子网；
- 防火墙；
- 路由表；
- 端口；
- 访问控制列表（ACL）；
- 访问管理；
- 网关；
- 网际协议（IP）基础，包括IP地址。

## 6.1 谁该对安全负责

AWS是一个责任共担的环境，就是说安全责任是由AWS和用户共同承担的。AWS承担以下责任。

- 通过自动监测系统和健壮的互联网访问来保护网络避免受到分布式拒绝服务（DDoS）攻击。
- 对能访问敏感区域的雇员进行背景调查。
- 存储设备退役时会在它们的生命周期结束后通过物理方式进行破坏。
- 确保数据中心的物理和环境安全，包括防火和保安人员。

这些安全标准由第三方团体审查。用户可以在AWS官方网站找到最新的概览。

用户承担以下责任。

- 加密网络数据传输来防止攻击者读取或操纵数据（如HTTPS）。
- 使用安全组和ACL来为自己的虚拟私有网络配置防火墙，以控制流入和流出的数据。
- 管理虚拟服务器上的操作系统及其他软件的补丁。
- 使用IAM实现访问管理来尽可能地限制对如S3或EC2这样的AWS资源的访问。

云中的安全关系到AWS和用户的相互影响。如果用户遵守规则，就能在云中达到高的安全标准。

## 6.2 使软件保持最新

每个星期都有修复安全漏洞的重要更新发布。有时候用户的操作系统会受到影响，有时候像OpenSSL这样的软件库会受到影响，有时候Java、Apache和PHP这样的环境会受到影响，有时候WordPress这样的应用程序会受到影响。如果一个安全更新发布了，用户必须快速安装它，因为如何利用这一漏洞可能与更新同时发布了，或者说每个人都能通过查看源代码来重建漏洞。用户应该有如何尽快将更新应用到所有正在运行的服务器上的工作计划。

### 6.2.1 检查安全更新

如果用户通过SSH登录到一台Amazon Linux EC2实例，就会看到下面当日的消息：

```
$ ssh ec2-user@ec2-52-6-25-163.compute-1.amazonaws.com
Last login: Sun Apr 19 07:08:08 2015 from [...]

  _ | _ | _ |
  _ | ( _ | /
  _ | \ _ | _ |

Amazon Linux AMI
https://aws.amazon.com/[...]/2015.03-release-notes/
4 package(s) needed for security, out of 28 available    <--4 个安全更新可用
Run "sudo yum update" to apply all updates.
```

这个例子表示4个安全更新可用，这个数目在更新的时候可能会发生变化。AWS不会替用户在其EC2实例上应用这些补丁——这么做是用户的责任。

用户可以使用yum程序包管理器在Amazon Linux上处理这些更新。运行**yum-security check-update** 来看哪些程序包需要安全更新：

```
$ yum --security check-update      <--读者运行这一命令时结果可能不同
4 package(s) needed for security, out of 28 available

[...]
openssl.x86_64                    1:1.0.1k-1.84.amzn1 amzn-updates      <--OpenSSL
是一个用于SSL 加密的库
[...]
unzip.x86_64                      6.0-2.9.amzn1      amzn-updates      <--unzip 可
以（解）压缩文件
[...]
```

我们鼓励用户追随Amazon Linux AMI Security Center获取影响Amazon Linux的安全公告。当一个新的安全更新发布时，用户应该检查自己是否受到影响。

在处理安全更新时，用户可能遇见下面两种状况之一。

- 当服务器第一次启动时，需要安装很多安全更新来使服务器保持最新。
- 新的安全更新发布时用户的服务器正在运行，用户需要在服务器运行时安装这些更新。

让我们来看看如何处理这些状况。

## 6.2.2 在服务器启动时安装安全更新

如果用户使用CloudFormation模板创建自己的EC2实例，用户可以有3种选项在启动时安装安全更新。

- 在服务器启动时安装所有更新。在自己的用户数据脚本中加入 **yum-y update**。
- 在服务器启动时仅安装安全更新。在自己的用户数据脚本中加入 **yum-y-security update**。

- 明确指定程序包版本 。安装指定版本号的更新。

前两个选项很容易被添加到用户的EC2实例的用户数据中。用户可以安装所有更新，如下所示：

```
[...]
"Server": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    [...]
    "UserData": {"Fn::Base64": {"Fn::Join": ["", [
      "#!/bin/bash -ex\n",
      "yum -y update\n"      <--服务器启动时安装所有更新
    ]]]}
  }
}
[...]
```

仅安装安全更新的话可以这样做：

```
[...]
"Server": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    [...]
    "UserData": {"Fn::Base64": {"Fn::Join": ["", [
      "#!/bin/bash -ex\n",
      "yum -y --security update\n"      <--服务器启动时仅安装安全更新
    ]]]}
  }
}
[...]
```

安装所有更新的问题在于用户的系统变得难以预料。如果用户的服务器是上一周启动的，所有被应用的更新是上一周发布的。然而同时，新的更新被发布了。如果用户今天启动一台新服务器并且安装所有更新，那么用户最终将得到一台与上一周完全不同的服务器。不同意味着因为某些原因它可能不再工作。这就是我们鼓励用户明确定义自己想安装的更新的原因。要明确指定所安装的安全更新的版本，用户可以使

用`yum update-to` 命令。`yum update-to` 使用明确的版本来更新程序包，而不是使用最新的：

```
yum update-to openssl-1.0.1k-1.84.amzn1 \      <--更新openssl 至版本1.0.1k-1.84.amzn1
unzip-6.0-2.9.amzn1      <--更新unzip至版本6.0-2.9.amzn1
```

使用CloudFormation模板可以这样来描述明确指定更新的一个EC2实例：

```
[...]
"Server": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    [...]
    "UserData": {"Fn::Base64": {"Fn::Join": ["", [
      "#!/bin/bash -ex\n",
      "yum -y update-to openssl-1.0.1k-1.84.amzn1 unzip-6.0-2.9.amzn1\n"
    ]]]}
  }
}
[...]
```

同样的方法也适用于非安全相关的程序包更新。当有新的安全更新发布时，用户应该检查自己是否受到影响并修改用户数据来保持新系统的安全。

### 6.2.3 在服务器运行时安装安全更新

不时地，用户必须在自己所有正在运行的服务器上安装安全更新。用户可以手动使用SSH登录到自己所有的服务器上，然后运行`yum-y-security update` 或者`yum update-to[...]`，但是如果用户有很多服务器或服务器数目在增长，这会变得很烦人。一个办法是使用一个小脚本获取自己的服务器列表，然后在所有服务器上运行`yum`来使这个任务自动化。代码清单6-1展示了在Bash中怎样能做到。读者可以在本书



的代码目录/chapter6/update.sh中找到相应的代码。

代码清单6-1 在所有正在运行的EC2实例上安装安全更新

```
PUBLICNAMES=$(aws ec2 describe-instances \      <--获得所有正在运行的EC2实例
的公共DNS 名
--filters "Name=instance-state-name,Values=running" \
--query "Reservations[].Instances[].PublicDnsName" \
--output text)

for PUBLICNAME in $PUBLICNAMES; do
    ssh -t -o StrictHostKeyChecking=no ec2-user@$PUBLICNAME \      <--通过SSH
    连接.....

    "sudo yum -y --security update"      <--.....然后执行yum update
done
```

现在我们可以快速地将更新应用到所有正在运行的服务器上了。

有些安全更新要求重启虚拟服务器。例如，如果用户需要给自己的运行Linux的虚拟服务器内核打补丁。用户可以自动重启服务器或切换到一个已经更新的AMI，然后启动一台新的虚拟服务器。例如，一个新的Amazon Linux AMI一年发布4次。

## 6.3 保护AWS账户安全

保护AWS账户安全非常关键。如果有人能访问你的AWS账户，他们就可以窃取你的数据，破坏任意东西（如数据、备份、服务器），或者窃取你的身份来干坏事。图6-1展示了一个AWS账户。每个AWS账户有一个root用户。在本书的例子中，当我们使用管理控制台时就是使用root用户；如果使用CLI，我们将使用4.2节中创建的用户mycli。除root用户之外，一个AWS账户中包含了你拥有的所有资源，如EC2实例、CloudFormation堆栈、IAM用户等。

要访问你的AWS账户，攻击者必须能使用你的账户认证。有3种方法可以做到这一点：使用root用户，使用一个普通用户，或者被认证为一个AWS资源（如EC2实例）。要被认证为一个（根）用户，攻击者需要密码或访问密钥。要认证为一个AWS资源（如EC2服务器），攻击者需要那个EC2实例发送API/CLI请求。

在本节中，我们将开始使用多重身份验证（MFA）来保护你的root用户。然后你将停止使用root用户，创建一个新用户用于日常操作，然后学会授予一个角色最小权限。

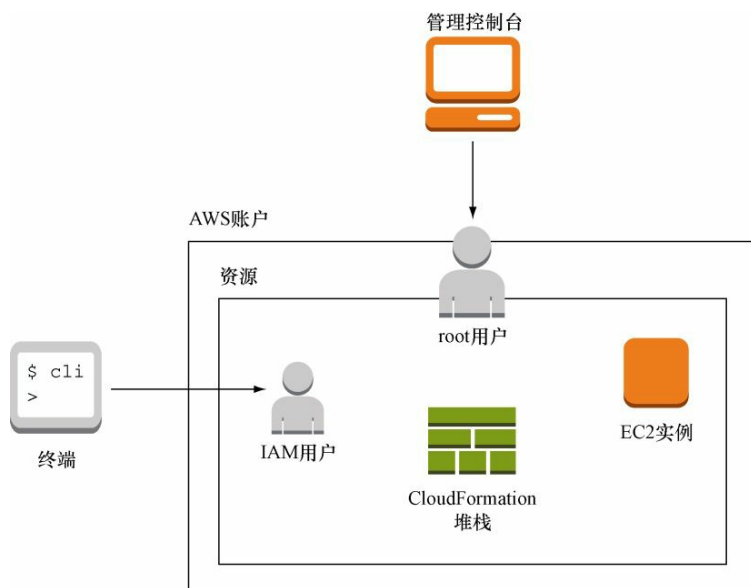


图6-1 一个AWS账户包含所有AWS资源，并且默认有一个root用户

## 6.3.1 保护AWS账户的root用户安全

如果你准备在产品中使用AWS，建议你为自己的root用户启用多重身份验证（MFA）。在MFA被激活后，你需要一个密码和一个临时令牌来作为root用户登录。这样一个攻击者不仅需要你的密码，还需要你的MFA设备。

按照下面的步骤来启用MFA，如图6-2所示。

- （1）在管理控制台的导航栏顶部点击你的名字。
- （2）点击“安全凭证”。
- （3）第一次可能会出现一个弹出框，需要选择继续安全凭证。
- （4）在智能手机上安装一个MFA应用（如Google Authenticator）。
- （5）展开多重身份验证（MFA）。
- （6）点击激活MFA。
- （7）跟随向导中的指令。使用智能手机上的MFA应用扫描所显示的QR码。

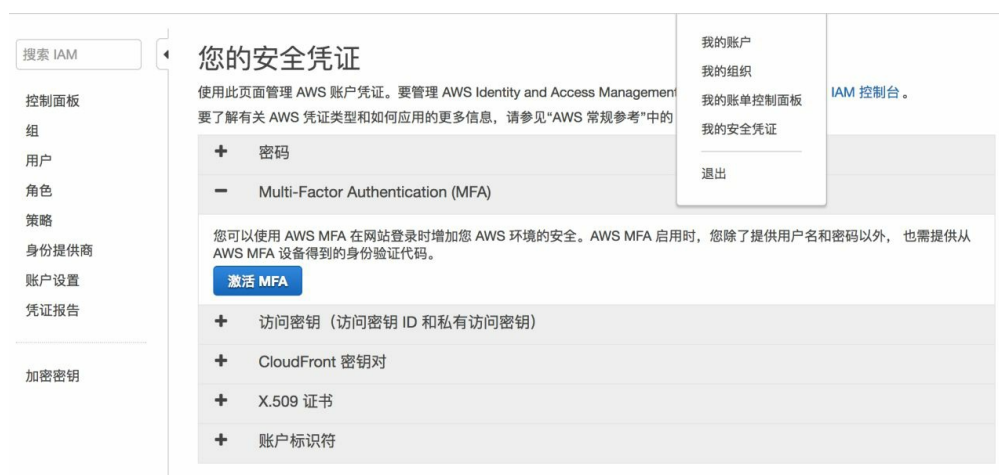


图6-2 使用多重身份验证（MFA）保护你的root用户

如果你使用自己的智能手机作为一个虚拟MFA设备，一个好主意是  
不用从自己的智能手机登录管理控制台或者在手机上存储root用户的密  
码。确保MFA令牌与自己的密码分离。

### 6.3.2 IAM服务

IAM（Identity and Access Management）服务通过AWS API提供身  
份认证与授权所需要的一切。你通过AWS API发送的每个请求都会通过  
IAM来检查这个请求是否被允许。IAM控制在你的AWS账户里，谁（身  
份认证）能做什么（授权）：谁被允许创建EC2实例？用户是否被允许  
终止一个特定的EC2实例？

使用IAM进行身份认证是通过用户和角色完成的，而授权是通过策  
略完成的。表6-1展示了用户和角色的区别。角色认证一个EC2实例，而  
用户可以用于其他一切。

表6-1 root用户、IAM用户和IAM角色的区别

	root用户	IAM用户	IAM角色
可以有一个密码	总是	是	否
可以有一个访问密钥	是（不推荐）	是	否
可以属于一个组	否	是	否
可以与一个EC2实例关联	否	否	是

IAM用户和IAM角色使用策略进行授权。在我们继续讲解用户和角  
色时首先看一下策略。记住用户和角色不能做任何事直到你在策略中允  
许特定的操作。

### 6.3.3 用于授权的策略

每个策略在JSON中定义且包含一个或多个声明。一个声明可以允许或拒绝在特定资源上做特定操作。用户可以在AWS官方网站找到可用于EC2资源的所有操作的概览。通配符\* 可以被用来创建更通用的声明。

下面的策略有一个声明允许对EC2服务中所有资源进行任意操作：

```
{
  "Version": "2012-10-17",      <--指定2012-10-17 来锁定版本
  "Statement": [{
    "Sid": "1",
    "Effect": "Allow",          <--允许（另一个选项是拒绝）.....
    "Action": ["ec2:*"],        <--.....所有EC2 操作（通配符*）.....
    "Resource": ["*"]          <--.....和所有资源
  }]
}
```

如果在同一操作上有多个声明，拒绝将覆盖允许。下面的策略允许除了终止实例以外的所有EC2操作：

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "1",
    "Effect": "Allow",
    "Action": ["ec2:*"],
    "Resource": ["*"]
  }, {
    "Sid": "2",
    "Effect": "Deny",          <--拒绝.....
    "Action": ["ec2:TerminateInstances"],    <--.....终止EC2 实例
    "Resource": ["*"]
  }]
}
```

下面的策略拒绝所有EC2操作。声明`ec2:TerminateInstances`不是关键性的，因为`Deny`覆盖`Allow`。当你拒绝一个操作时，是无法通过在另一个声明来允许它的：

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "1",
    "Effect": "Deny",
    "Action": ["ec2:*"],      <--拒绝所有EC2 操作
    "Resource": ["*"]
  }, {
    "Sid": "2",
    "Effect": "Allow",
    "Action": ["ec2:TerminateInstances"],    <--允许不是关键性的，拒绝覆盖允许
    "Resource": ["*"]
  }]
}
```

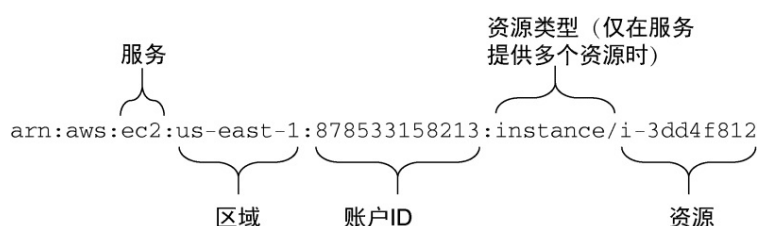


图6-3 用于标识一个EC2实例的Amazon Resource Name（ARN）的组成

至此，**Resource** 部分我们使用了`["*"]` 来表示所有资源。在AWS中资源有一个Amazon Resource Name（ARN）；图6-3显示了一个EC2实例的ARN。要找出账户ID，你可以使用CLI：

```
$ aws iam get-user --query "User.Arn" --output text
arn:aws:iam::878533158213:user/mycli    <--账户ID 有12 个数字
```

如果你知道自己的账户ID，则可以使用ARN来允许访问一个服务的特定资源：

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "2",
    "Effect": "Allow",
    "Action": ["ec2:TerminateInstances"],
    "Resource": ["arn:aws:ec2:us-east-1:878533158213:instance/i-3dd4f812"]
  }]
}
```

有以下两种类型的策略。

- 托管策略 —— 如果用户想创建可以在自己的账户内复用的策略，托管策略正是用户要找的。有以下两种类型的托管策略。
  - AWS托管策略 —— AWS维护的策略。有一些授予管理员权限、只读权限等的策略。
  - 客户托管策略 —— 可以是代表你的组织中的角色的策略。
- 内联策略 —— 属于某个特定IAM角色、用户或组的策略。内联策略不能存在于IAM角色、用户或组之外。

通过**CloudFormation**，内联策略很容易维护；这就是为什么我们在本书中大多数时候使用内联策略的原因。一个例外是**mycli**用户：这个用户被附加了AWS托管策略**AdministratorAccess**。

## 6.3.4 用于身份认证的用户和用于组织用户的组

一个用户可以被密码或访问密钥所认证。当用户登录到管理控制台，用户是通过密码认证的。如果你在计算机中使用了CLI，就可以使用一个访问密钥来认证**mycli**用户。

目前用户正使用**root**用户登录管理控制台。因为使用最小权限总是个好主意，你将为管理控制台创建一个新用户。为了使将来添加用户更容易，首先要为所有管理员用户创建一个组。一个组不能被用来作为身份认证，但它集中了授权。如果想阻止管理员用户终止EC2服务器，只需修改这个组的策略，而不是所有管理员用户的策略。一个用户可以不是任何组的成员，也可以是一个或多个组的成员。

使用CLI很容易创建组和用户。使用一个安全的密码来替换\$Password：

```
$ aws iam create-group --group-name "admin"
$ aws iam attach-group-policy --group-name "admin" \
--policy-arn "arn:aws:iam::aws:policy/AdministratorAccess"
$ aws iam create-user --user-name "myuser"
$ aws iam add-user-to-group --group-name "admin" --user-name "myuser"
$ aws iam create-login-profile --user-name "myuser" --password "$Password"
```

用户myuser已经准备好可以使用了。但是，如果不是使用root用户的话，则必须使用一个不同的URL来访问管理控制台，即[https://\\$accountId.signin.aws.amazon.com/console](https://$accountId.signin.aws.amazon.com/console)。用之前使用aws iam get-user 提取的账户ID替换\$accountId。

#### 为IAM用户启用MFA

我们也鼓励用户为所有用户启用MFA。如果可能，不要为自己的root用户和日常用户使用同样的MFA设备。用户可以从AWS合作伙伴，如Gemalto那里以13美元购买硬件MFA设备。按下列步骤为自己的用户启用MFA。

- (1) 在管理控制台中打开IAM服务。
- (2) 在左侧选择用户。
- (3) 选择用户myuser。
- (4) 在页面底部的Sign-In Credentials部分中点击Manage MFA Device按钮。向导界面和root用户是相同的。

应该为所有拥有密码的用户激活MFA，即哪些可以使用管理控制台的用户。

#### 警告

从现在起停止使用root用户。总是使用myuser及管理控制台的新链接。

#### 警告

用户永远不应该把一个用户访问密钥复制到一个EC2实例上，要使用IAM角色！不要在自己的源代码中存储安全凭证，并且永远不要把它们提交到自己的Git或SVN资源库中。替代



方案是，在可能的情况下尝试使用IAM角色。

## 6.3.5 用于认证AWS的角色

IAM角色可以被用来认证AWS资源，如虚拟服务器。可以不为一个EC2实例附加角色，也可以附加一个或多个角色。从一个AWS资源（如EC2实例）发送的每个AWS API请求都会使用附加的角色进行认证。如果AWS资源附加了一个或多个角色，IAM会检查这些角色附加的所有策略来确定请求是否是被允许的。默认情况下，EC2实例没有任何角色，所以就不被允许调用任何AWS API。

还记得第4章中的临时EC2实例吗？如果临时服务器没有被终止——人们经常忘了这么做。许多钱就这样被浪费了。现在我们将创建一个过一会儿会自己停止的EC2实例。**at** 命令将在5 min延迟后停止这个实例：

```
echo "aws ec2 stop-instances --instance-ids i-3dd4f812" | at now + 5 minutes
```

这个EC2实例需要停止自己的授权。我们可以使用内联策略来进行授权。下面的代码展示了如何在CloudFormation中把角色定义成一个资源：

```
"Role": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {      <--- 魔术：复制和粘贴
      "Version": "2012-10-17",
      "Statement": [{
        "Effect": "Allow",
        "Principal": {
          "Service": ["ec2.amazonaws.com"]
        },
        "Action": ["sts:AssumeRole"]
      }]
    },
    "Path": "/",
```

```

"Policies": [{      <--策略开始
  "PolicyName": "ec2",
  "PolicyDocument": {      <--策略定义
    "Version": "2012-10-17",
    "Statement": [{

      "Sid": "Stmt1425388787000",
      "Effect": "Allow",
      "Action": ["ec2:StopInstances"],
      "Resource": ["*"],      <--角色之后创建EC2 实例：不能{"Ref"}一个实例ID!

      "Condition": {      <--Condition 可以解决这一问题：只允许用堆栈ID 打上标签的
        "StringEquals": {"ec2:ResourceTag/aws:cloudformation:stack-id":
          {"Ref": "AWS::StackId"}}
      }
    }
  ]
}
}]
}
}
}
}

```

要附加一个角色给实例，必须首先创建一个实例配置文件：

```

"InstanceProfile": {
  "Type": "AWS::IAM::InstanceProfile",
  "Properties": {
    "Path": "/",
    "Roles": [{"Ref": "Role"}]
  }
}

```

现在可以把角色和EC2实例关联起来了：

```

"Server": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    "IamInstanceProfile": {"Ref": "InstanceProfile"},
    [...],
    "UserData": {"Fn::Base64": {"Fn::Join": ["", [
      "#!/bin/bash -ex\n",

```

```
    "INSTANCEID=`curl -s `,  
    "http://169.254.169.254/latest/meta-data/instance-id`\n",  
    "echo \"aws --region us-east-1 ec2 stop-instances ",  
    "--instance-ids $INSTANCEID\" | at now + 5 minutes\n"  
  ]]]}  
}  
}
```

使用位于<https://s3.amazonaws.com/awsinaction/chapter6/server.json>的模板创建CloudFormation堆栈。我们可以通过参数指定服务器的生命周期。等待生命周期结束，然后看看实例是不是被停止了。生命周期在服务器完全启动和引导后开始。

#### 资源清理

在本节结束时别忘了删除堆栈来清除所有用过的资源，否则很可能会因为使用这些资源被收取费用。

## 6.4 控制进出虚拟服务器的网络流量

用户只希望必要的数据流量进出自己的EC2实例。使用防火墙，用户可以控制进入（也叫作inbound或ingress）和出去（也叫作outbound或egress）的数据流量。如果用户运行一台网站服务器，用户需要对外面的世界打开的端口只有HTTP流量使用的80端口及HTTPS流量使用的443端口。所有其他端口都应该被关闭，只打开必要的端口，就像只通过IAM授予最小的权限那样。如果用户有一个严格的防火墙，就关闭了许多可能的安全漏洞。用户也可以通过不为测试系统开放出去的SMTP连接来阻止不小心从测试系统发送给客户的邮件。

在网络流量进入或离开用户的EC2实例之前，它将穿过AWS提供的防火墙。这个防火墙审查网络流量并使用规则来确定流量是被允许的还是被拒绝的。

### IP和IP地址

缩写IP代表Internet Protocol（网际协议），而IP地址则类似于84.186.116.47。

图6-4展示了一个来自源IP地址10.0.0.10的SSH请求是怎样被防火墙检查，然后被目的地IP地址10.10.0.20收到的。在这一案例中，防火墙允许这一请求因为有条规则允许源和目的地址之间的端口22上的TCP流量。

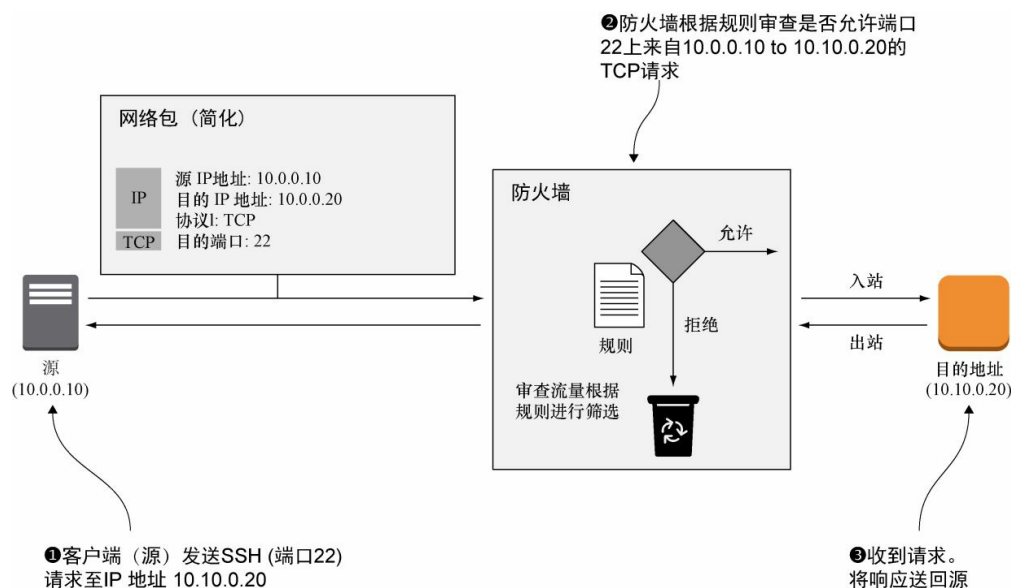


图6-4 一个SSH请求如何从源到目的地，并受防火墙控制

#### 源和目的地

入站安全组规则筛选基于网络流量的源。源可以是一个IP地址也可以是一个安全组。这样就可以只允许从特定源IP地址范围来的入站流量。

出站安全组规则筛选基于网路流量的目的地。目的地可以是一个IP地址或安全组。可以只允许特定目的IP地址范围的出站流量。

AWS对防火墙负责，但你对规则负责。默认情况下，所有的入站流量都被拒绝而所有的出站流量都被允许。然后你可以开始允许入站流量。如果开始添加出站流量规则，则默认值会从允许所有切换至拒绝所有，只有你添加的例外会被允许。

### 6.4.1 使用安全组控制虚拟服务器的流量

一个安全组可以被关联到AWS资源（如EC2实例）。通常EC2实例有超过一个安全组与之关联，而同一安全组会被关联到许多个EC2实例。

一个安全组遵循一组规则。一个规则可以基于以下内容允许网络流量：

- 方向（入站或出站）；
- IP协议（TCP、UDP、ICMP）；
- 源IP地址/目的IP地址；
- 端口；
- 源安全组/目的安全组（仅在AWS上有效）。

你可以定义规则来允许所有的流量进入和离开你的服务器，AWS不会阻止你这么。但是一种好的做法是定义规则，使它们尽可能严格。

在CloudFormation里一个安全组资源类型是**AWS::EC2::SecurityGroup**。代码清单6-2在本书的代码目录/chapter6/firewall1.json中。这个模板描述了一个与单个EC2实例关联的空的安全组。

代码清单6-2 关联单个EC2实例的空安全组

```
{
  "Parameters": {
    "KeyName": {
      "Type": "AWS::EC2::KeyPair::KeyName",
      "Default": "mykey"
    },
    "VPC": {      <--6.5 节将介绍相关内容
      [...]
    },
    "Subnet": {   <--6.5 节将介绍相关内容
      [...]
    }
  },
  "Resources": {
    "SecurityGroup": {    <--安全组描述
      "Type": "AWS::EC2::SecurityGroup",
      "Properties": {
        "GroupDescription": "My security group",
        "VpcId": {"Ref": "VPC"}
      }
    },
    "Server": {          <--EC2 实例的描述
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "ImageId": "ami-1ecae776",
        "InstanceType": "t2.micro",
```

```

        "KeyName": {"Ref": "KeyName"},
        "SecurityGroupIds": [{"Ref": "SecurityGroup"}],    <--用Ref 将安全
组关联到EC2 实例
        "SubnetId": {"Ref": "Subnet"}
    }
}
}
}

```

要探究安全组，可以在位于 <https://s3.amazonaws.com/awsinaction/chapter6/firewall1.json> 的 CloudFormation 模板上尝试。基于这个模板创建一个堆栈，然后从堆栈输出复制 `PublicName`。

## 6.4.2 允许ICMP流量

如果用户要从自己的计算机 ping 一个 EC2 实例，就必须允许来自因特网控制报文协议（Internet Control Message Protocol, ICMP）流量入站。默认情况下，所有的入站流量都被阻止了。尝试 ping `$PublicName` 来确认 ping 访问不被允许：

```

$ ping ec2-52-5-109-147.compute-1.amazonaws.com
PING ec2-52-5-109-147.compute-1.amazonaws.com (52.5.109.147): 56 data byte
s
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
[...]

```

用户需要在安全组中添加一条规则来允许入站流量，其中协议是 ICMP。代码清单 6-3 可以在本书的代码目录 `/chapter6/firewall2.json` 中找到。

代码清单 6-3 允许 ICMP 的安全组

```

{
  [...]

```

```

"Resources": {
  "SecurityGroup": {
    [...]
  },
  "AllowInboundICMP": {      <-- 允许ICMP 规则描述
    "Type": "AWS::EC2::SecurityGroupIngress",      <-- 入站规则类型
    "Properties": {
      "GroupId": {"Ref": "SecurityGroup"},      <-- 将规则与安全组联系起来
      "IpProtocol": "icmp",      <-- 指定协议
      "FromPort": "-1",      <-- -1 意味着所有端口
      "ToPort": "-1",
      "CidrIp": "0.0.0.0/0"      <-- 0.0.0.0/0 意味着所有源IP 地址都被允许
    }
  },
  "Server": {
    [...]
  }
}
}

```

使用位于<https://s3.amazonaws.com/awsinaction/chapter6/firewall2.json>的模板更新CloudFormation堆栈，然后再次尝试ping 命令。现在它看上去应该是这样的：

```

$ ping ec2-52-5-109-147.compute-1.amazonaws.com
PING ec2-52-5-109-147.compute-1.amazonaws.com (52.5.109.147): 56 data byte
s
64 bytes from 52.5.109.147: icmp_seq=0 ttl=49 time=112.222 ms
64 bytes from 52.5.109.147: icmp_seq=1 ttl=49 time=121.893 ms
[...]
round-trip min/avg/max/stddev = 112.222/117.058/121.893/4.835 ms

```

现在每个人的入站ICMP流量（每个源IP地址）被允许抵达EC2实例。

## 6.4.3 允许SSH流量

当能够ping自己的EC2实例之后，用户会想要通过SSH登录自己的



服务器。要这么做，必须创建一条规则，允许端口22上的入站TCP请求，如代码清单6-4所示。

代码清单6-4 允许SSH的安全组

```
[...]
"AllowInboundSSH": {      <-- 允许SSH规则描述
  "Type": "AWS::EC2::SecurityGroupIngress",
  "Properties": {
    "GroupId": {"Ref": "SecurityGroup"},
    "IpProtocol": "tcp",    <-- SSH 基于TCP 协议
    "FromPort": "22",      <-- 默认的SSH 端口是22
    "ToPort": "22",        <-- 允许一个范围的端口或者设置FromPort = ToPort
    "CidrIp": "0.0.0.0/0"
  }
},
[...]
```

使用位于<https://s3.amazonaws.com/awsinaction/chapter6/firewall3.json>的模板更新CloudFormation堆栈。现在我们可以使用SSH登录自己的服务器了。记住，我们还需要正确的私钥。防火墙只是控制网络层，它不能替代基于密钥或密码的身份认证。

## 6.4.4 允许来自源IP地址的SSH流量

目前为止，我们允许从任意IP地址来的端口22（SSH）上的入站流量。

在模板中硬编码进公有IP地址并不是一个好的解决方案，因为它会时不时发生变化。但是我们已经知道解决方法了——参数。我们需要添加一个参数来保存自己当前的公有IP地址，然后需要修改AllowInboundSSH 规则。读者可以在本书的代码目录/chapter6/firewall4.json中找到代码清单6-5。

代码清单6-5 仅允许从特定IP地址来的SSH的安全组

```
[...]
"Parameters": {
```

```
[...]
"IpForSSH": {      <-- 公有IP 地址参数
  "Description": "Your public IP address to allow SSH access",
  "Type": "String"
},
},
"Resources": {
  "AllowInboundSSH": {
    "Type": "AWS::EC2::SecurityGroupIngress",
    "Properties": {
      "GroupId": {"Ref": "SecurityGroup"},
      "IpProtocol": "tcp",
      "FromPort": "22",
      "ToPort": "22",
      "CidrIp": {"Fn::Join": ["", [{"Ref": "IpForSSH"}, "/32"]]}      <-- 使
用$IpForSSH/32 作为值
    }
  },
  [...]
}
```

#### 公有IP地址和私有IP地址有什么区别

在我的本地网络，我是使用以192.168.0.\*开头的私有IP地址。我的笔记本电脑使用192.168.0.10，而我的iPad使用192.168.0.20。但是，如果我访问互联网，我的笔记本电脑和iPad使用相同的公有IP（如79.241.98.155）。这是因为只有我的互联网网关（那个连接至互联网的盒子）有一个公有IP地址，而所有的请求都被网关重定向（如果读者想深入研究这点，可查找网络地址转换）。你的本地网络并不知道这个公有IP地址。我的笔记本电脑和iPad只知道互联网网关可以在私有网络的192.168.0.1访问到。

要找出你的公有IP地址，访问<http://api.ipify.org>。对大多数人来说，通常当我们重新连接到互联网时（在我的案例中大约每24小时发生一次），公有IP地址会时不时地发生变化。

使用位于<https://s3.amazonaws.com/awsinaction/chapter6/firewall4.json>的模板更新CloudFormation堆栈。当要求输入参数时，输入你的公有IP地址\$IPForSSH。现在只有你的IP地址能够建立到你的EC2实例的SSH连接。

#### 无类别域间路由（CIDR）

读者可能想知道代码清单6-5中的/32 是什么意思。要理解发生了什么，需要将思维切换至二进制模式。一个IP地址的长度是4字节（即32位）。/32 定义的位数（在这一案例中是

32) 应该被用来组成一个地址范围。如果想定义被允许的准确的IP地址，必须使用32位。

但是，有时候定义一个允许的IP地址范围是合理的。例如，我们可以使用`10.0.0.0/8`来创建一个10.0.0.0至10.255.255.255之间的范围，使用`10.0.0.0/16`来创建一个10.0.0.0至10.0.255.255之间的范围，或者使用`10.0.0.0/24`来创建一个10.0.0.0至10.0.0.255之间的范围。不是必须使用二进制边界（8, 16, 24, 32），只是它们对大多数人来说更容易理解。我们已经使用了`0.0.0.0/0`来创建一个包含每一个可能IP地址的范围。

现在我们已经能够通过根据协议、端口和源IP地址做筛选，来控制从AWS外部流入或流出至AWS外部的流量了。

## 6.4.5 允许来自源安全组的SSH流量

如果要控制从一个AWS资源（如一个EC2实例）到另一个AWS资源的流量，安全组是很强大的。可以根据源与目的地是否属于一个特定的安全组来控制网络流量。例如，我们可以定义一个MySQL数据库只能被自己的网站服务器访问，或只有我们的网络缓存服务器被允许访问网站服务器。因为云的弹性性质，用户很可能要处理动态数量的服务器，所以基于源IP地址的规则难以维护。如果用户的规则是基于源安全组的，这就会变得很容易。

要探究基于源安全组的规则的能力，我们来看一下用于SSH访问的堡垒主机（有些人称其为跳转盒）的概念。其中的技巧在于只有一台服务器——堡垒主机，能通过SSH被互联网访问（应该被限制到一个特定的源IP地址）。所有其他服务器只能从堡垒主机使用SSH访问。这一方法有两个优势。

- 用户的系统只有一个入口，且这一入口除了SSH不做其他事。这个盒子被攻破的机会很小。
- 如果用户的某台网站服务器、邮件服务器、FTP服务器等被攻破了，攻击者无法从这台服务器跳转到所有其他服务器。

要实现堡垒主机的概念，必须遵守这两条规则。

- 允许从`0.0.0.0/0`或一个指定的源地址使用SSH访问堡垒主机。
- 只有当流量来源于堡垒主机时才允许使用SSH访问所有其他服务器。

图6-5展示了一台堡垒机加两台服务器的体系。这两台服务器仅仅接受来自堡垒机的入站SSH访问。

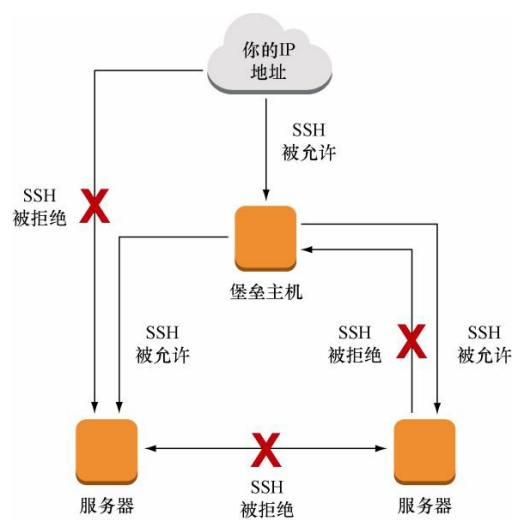


图6-5 堡垒主机是唯一能用SSH访问系统的点，从它可以通过SSH访问所有其他服务器（使用安全组实现）

代码清单6-6展示了允许从特定源安全组的SSH规则。

代码清单6-6 Se允许从堡垒主机使用SSH访问的安全组

```
[...]
"SecurityGroupPrivate": {      <---新安全组
  "Type": "AWS::EC2::SecurityGroup",
  "Properties": {
    "GroupDescription": "My security group",
    "VpcId": {"Ref": "VPC"}
  }
},
"AllowPrivateInboundSSH": {
  "Type": "AWS::EC2::SecurityGroupIngress",
  "Properties": {
    "GroupId": {"Ref": "SecurityGroupPrivate"},
    "IpProtocol": "tcp",
    "FromPort": "22",
    "ToPort": "22",
    "SourceSecurityGroupId": {"Ref": "SecurityGroup"}      <---仅当源是其他安全组时允许
  }
},
[...]
```

使用位于<https://s3.amazonaws.com/awsinaction/chapter6/firewall5.json>的模板更新CloudFormation堆栈。如果更新完成了，堆栈会显示以下3个输出。

- **BastionHostPublicName** ——从你的计算机通过SSH使用堡垒主机连接。
- **Server1PublicName** ——你只能从堡垒主机连接这台服务器。
- **Server2PublicName** ——你只能从堡垒主机连接这台服务器。

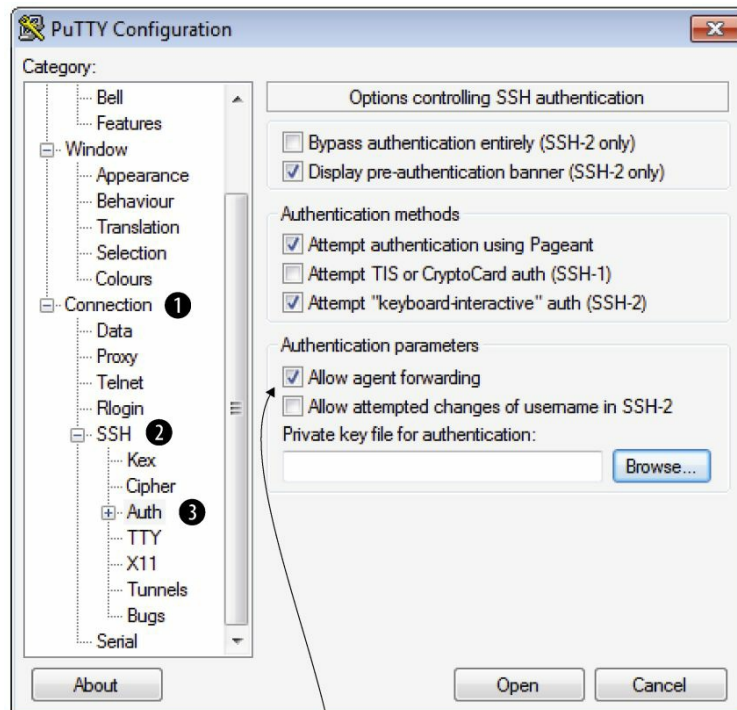
现在通过SSH命令`ssh -I $PathToKey/mykey.pem -A ec2-user@$BastionHost` ``PublicName` 连接**BastionHostPublicName**。把**\$PathToKey** 替换成你的SSH密钥路径，把**\$BastionHostPublicName** 替换成你的堡垒主机的公开名称。**-A** 选项用于启用**AgentForwarding**，代理转发让你可以使用你登录到堡垒主机的同一密钥来为进一步的从堡垒主机发起的SSH登录进行身份认证。

执行下面的命令来把你的密钥加到SSH代理。使用SSH密钥路径替换**\$PathToKey**：

```
ssh-add $PathToKey/mykey.pem
```

## 6.4.6 用PuTTY进行代理转发

要使用PuTTY进行代理转发，我们需要确保已经双击私钥文件将密钥装载到PuTTY Pageant。同时必须启用  
Connection → SSH → Auth → Allow Agent Forwarding，如图6-6所示。



启用代理转发

图6-6 通过PuTTY允许代理转发

这样从堡垒主机就可以继续登录到`$Server1PublicName` 或者`$Server2PublicName`：

```
[computer]$ ssh -i mykey.pem -A ec2-user@ec2-52-4-234-102.[...].com    <--
-登录到堡垒主机
Last login: Sat Apr 11 11:28:31 2015 from [...]
[...]
[bastionh]$ ssh ec2-52-4-125-194.compute-1.amazonaws.com    <--从堡垒主机
登录到$Server1PublicName
Last login: Sat Apr 11 11:28:43 2015 from [...]
[...]
```

堡垒主机可以用来为系统增加一层安全保护。如果其中一台服务器被攻陷了，攻击者无法跳转到系统中的其他服务器上。这样减少了一个攻击者能够造成的潜在的损害。堡垒主机只做SSH不做其他事，这一点很重要，这样能减少它成为安全风险的机会。我们经常使用堡垒主机模式来保护我们的客户。

### 资源清理

在本节结束时别忘了删除堆栈来清除所有用过的资源，否则很可能会因为使用这些资源被收取费用。

## 6.5 在云中创建一个私有网络：虚拟私有云

通过创建一个虚拟私有云（virtual private cloud, VPC），用户将在AWS上得到自己的私有网络。私有意味着用户可以使用地址范围10.0.0.0/8、172.16.0.0/12或192.168.0.0/16来设计一个网络，它不一定要连接到公有互联网。用户可以创建子网、路由表、访问控制列表（ACL）以及访问互联网的网关或VPN端点。

一个子网可以让用户分离关注点。为用户的数据库、网站服务器、缓存服务器或应用服务器，或者任何能分离的两个系统创建新的子网。另一条经验是用户应该至少有两个子网，即公有子网和私有子网。公有子网能够路由到互联网，私有子网则不能。用户的网站服务器应该在公有子网中，而用户的数据库应该在私有子网中。

为了理解VPC是如何工作的，我们将创建一个VPC来放置一个企业网站应用。我们将通过创建一个只包含堡垒主机服务器的公有子网重新实现6.4节中的堡垒主机概念。同时我们将为网站服务器创建一个私有子网且为网站缓存创建一个公有子网。网站缓存将通过返回在缓存中的最新版本页面来吸收大多数流量，并且将流量重定向到私有网站服务器。不能直接从互联网访问网站服务器——只能通过网络缓存。

这一VPC使用地址空间10.0.0.0/16。为了分离关注点，我们将在这个VPC中创建两个公有子网和一个私有子网：

- 10.0.1.0/24公有SSH堡垒主机子网；
- 10.0.2.0/24公有Varnish网络缓存子网；
- 10.0.3.0/24私有Apache网站服务器子网。

### 10.0.0.0/16是什么意思

10.0.0.0/16表示所有10.0.0.0至10.0.255.255之间的IP地址。它使用了CIDR标记法（本章前面介绍过）。

网络ACL像防火墙那样限制流量从一个子网流向另一个子网。6.5节中的SSH堡垒主机可以使用下面这些ACL来实现。



- 从0.0.0.0/0到10.0.1.0/24的SSH是被允许的。
- 从10.0.1.0/24到10.0.2.0/24的SSH是被允许的。
- 从10.0.1.0/24到10.0.3.0/24的SSH是被允许的。

要允许网络流量到Varnish网络缓存及HTTP服务器，需要额外的ACL。

- 从0.0.0.0/0到10.0.2.0/24的HTTP是被允许的。
- 从10.0.2.0/24到10.0.3.0/24的HTTP是被允许的。

图6-7展示了这一VPC的架构。

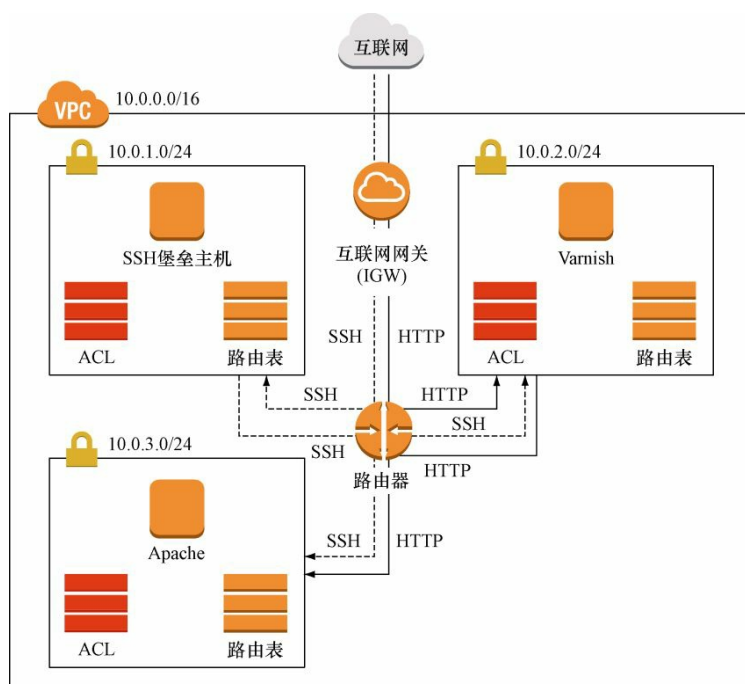


图6-7 有3个子网来保护网站应用安全的VPC

我们将使用CloudFormation来描述VPC和内部的子网。在本书中为了使其更好地被理解，这一模板被分割成若干小块。读者同样可以在本书的源代码中找到相应的代码。这个模板位于/chapter6/vpc.json。

## 6.5.1 创建VPC和IGW

这个模板中的第一个资源是VPC和IGW（Internet Gateway）。IGW

会使用网络地址转换（network address translation, NAT）把虚拟服务器的公有IP地址转换成它们的私有IP地址。这个VPC里的所有公有IP地址都由这个IGW控制：

```
"VPC": {
  "Type": "AWS::EC2::VPC",
  "Properties": {
    "CidrBlock": "10.0.0.0/16",      <--地址空间
    "EnableDnsHostnames": "true"
  }
},
"InternetGateway": {      <--通过IGW访问互联网并为公有IP 地址做NAT 转换
  "Type": "AWS::EC2::InternetGateway",
  "Properties": {
    [...]
  }
},
"VPCGatewayAttachment": {      <--将网关关联到VPC
  "Type": "AWS::EC2::VPCGatewayAttachment",
  "Properties": {
    "VpcId": {"Ref": "VPC"},
    "InternetGatewayId": {"Ref": "InternetGateway"}
  }
},
```

接下来我们将为堡垒主机定义子网。

## 6.5.2 定义公有堡垒主机子网

堡垒主机子网只有一台机器，以保护SSH访问安全：

```
"SubnetPublicSSHBastion": {
  "Type": "AWS::EC2::Subnet",
  "Properties": {
    "AvailabilityZone": "us-east-1a",      <--第11 章中将学习
    "CidrBlock": "10.0.1.0/24",          <--地址空间
    "VpcId": {"Ref": "VPC"}
  }
},
"RouteTablePublicSSHBastion": {      <--路由表
```

```

    "Type": "AWS::EC2::RouteTable",
    "Properties": {
        "VpcId": {"Ref": "VPC"}
    }
},
"RouteTableAssociationPublicSSHBastion": {    <--将路由表关联到子网
    "Type": "AWS::EC2::SubnetRouteTableAssociation",
    "Properties": {
        "SubnetId": {"Ref": "SubnetPublicSSHBastion"},
        "RouteTableId": {"Ref": "RouteTablePublicSSHBastion"}
    }
},
"RoutePublicSSHBastionToInternet": {
    "Type": "AWS::EC2::Route",
    "Properties": {
        "RouteTableId": {"Ref": "RouteTablePublicSSHBastion"},
        "DestinationCidrBlock": "0.0.0.0/0",    <--将任何地址（0.0.0.0/0）路由
至IGW
        "GatewayId": {"Ref": "InternetGateway"}
    },
    "DependsOn": "VPCGatewayAttachment"
},
"NetworkAclPublicSSHBastion": {    <--ACL
    "Type": "AWS::EC2::NetworkAcl",
    "Properties": {
        "VpcId": {"Ref": "VPC"}
    }
},
"SubnetNetworkAclAssociationPublicSSHBastion": {    <-- 将ACL 与子网关联
    "Type": "AWS::EC2::SubnetNetworkAclAssociation",
    "Properties": {
        "SubnetId": {"Ref": "SubnetPublicSSHBastion"},
        "NetworkAclId": {"Ref": "NetworkAclPublicSSHBastion"}
    }
},

```

ACL定义如下：

```

"NetworkAclEntryInPublicSSHBastionSSH": {    <--允许来自任何地方的入站SSH
    "Type": "AWS::EC2::NetworkAclEntry",
    "Properties": {
        "NetworkAclId": {"Ref": "NetworkAclPublicSSHBastion"},
        "RuleNumber": "100",
        "Protocol": "6",

```

```

    "PortRange": {
      "From": "22",
      "To": "22"
    },
    "RuleAction": "allow",
    "Egress": "false",      ←--入站
    "CidrBlock": "0.0.0.0/0"
  }
},
"NetworkAclEntryInPublicSSHBastionEphemeralPorts": {      ←--用于短TCP/IP 连接的临时端口
  "Type": "AWS::EC2::NetworkAclEntry",
  "Properties": {
    "NetworkAclId": {"Ref": "NetworkAclPublicSSHBastion"},
    "RuleNumber": "200",
    "Protocol": "6",
    "PortRange": {
      "From": "1024",
      "To": "65535"
    },
    "RuleAction": "allow",
    "Egress": "false",
    "CidrBlock": "10.0.0.0/16"
  }
},
"NetworkAclEntryOutPublicSSHBastionSSH": {      ←--允许从VPC出站的SSH
  "Type": "AWS::EC2::NetworkAclEntry",
  "Properties": {
    "NetworkAclId": {"Ref": "NetworkAclPublicSSHBastion"},
    "RuleNumber": "100",
    "Protocol": "6",
    "PortRange": {
      "From": "22",
      "To": "22"
    },
    "RuleAction": "allow",
    "Egress": "true",      ←--出站
    "CidrBlock": "10.0.0.0/16"
  }
},
"NetworkAclEntryOutPublicSSHBastionEphemeralPorts": {      ←--临时端口
  "Type": "AWS::EC2::NetworkAclEntry",
  "Properties": {
    "NetworkAclId": {"Ref": "NetworkAclPublicSSHBastion"},
    "RuleNumber": "200",
    "Protocol": "6",
    "PortRange": {

```

```

    "From": "1024",
    "To": "65535"
  },
  "RuleAction": "allow",
  "Egress": "true",
  "CidrBlock": "0.0.0.0/0"
}
},

```

安全组与ACL有一个重要的区别：安全组是有状态的，而ACL则没有。如果用户允许在安全组上的一个入站端口，那么该入站端口上的请求对应的出站响应也是被允许的。安全组规则将按用户所期望的方式工作。如果用户在安全组上打开端口22，就能通过SSH连接。

ACL则不同。如果用户仅仅为子网的ACL打开入站端口22，仍然不能通过SSH访问。除此之外，用户还需要允许出站临时端口，因为sshd（SSH守护进程）在端口22上接受连接，但却使用临时端口与客户端通信。临时端口从范围1024至65535中选择。

如果用户想从自己的子网发起一个SSH连接，就需要打开出站端口22且同时打开入站临时端口。如果对这些都不熟悉，应该使用安全组且在ACL层允许所有。

### 6.5.3 添加私有Apache网站服务器子网

Varnish网络缓存子网与堡垒主机子网类似，它也是一个公有子网。因此，我们跳过它，继续Apache网站服务器的私有子网：

```

"SubnetPrivateApache": {
  "Type": "AWS::EC2::Subnet",
  "Properties": {
    "AvailabilityZone": "us-east-1a",
    "CidrBlock": "10.0.3.0/24",      <---地址空间
    "VpcId": {"Ref": "VPC"}
  }
},
"RouteTablePrivateApache": {      <---没有路由到IGW
  "Type": "AWS::EC2::RouteTable",

```

```

    "Properties": {
      "VpcId": {"Ref": "VPC"}
    },
    "RouteTableAssociationPrivateApache": {
      "Type": "AWS::EC2::SubnetRouteTableAssociation",
      "Properties": {
        "SubnetId": {"Ref": "SubnetPrivateApache"},
        "RouteTableId": {"Ref": "RouteTablePrivateApache"}
      }
    },
  },

```

公有子网和私有子网唯一的区别是，私有子网不能路由到IGW。默认情况下，VPC内的子网之间的流量总是能被路由的。不能移除子网间的路由。如果想阻止VPC内部子网间的流量，需要使用与子网关联的ACL。

## 6.5.4 在子网中启动服务器

子网已经准备好了，我们可以继续操作EC2实例。首先，我们可以描述堡垒主机：

```

"BastionHost": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    "ImageId": "ami-1ecae776",
    "InstanceType": "t2.micro",
    "KeyName": {"Ref": "KeyName"},
    "NetworkInterfaces": [{
      "AssociatePublicIpAddress": "true",      <--- 分配一个公有IP 地址
      "DeleteOnTermination": "true",
      "SubnetId": {"Ref": "SubnetPublicSSHBastion"},      <--- 在堡垒主机子网中
    }],
    "DeviceIndex": "0",
    "GroupSet": [{"Ref": "SecurityGroup"}]      <--- 安全组允许所有
  }
},

```

Varnish服务器看上去与此类似。但是，私有Apache网站服务器的配置有所不同：

```
"ApacheServer": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    "ImageId": "ami-1ecae776",
    "InstanceType": "t2.micro",
    "KeyName": {"Ref": "KeyName"},
    "NetworkInterfaces": [{
      "AssociatePublicIpAddress": "false",      <---非公有IP 地址：私有的
      "DeleteOnTermination": "true",
      "SubnetId": {"Ref": "SubnetPrivateApache"},  <---在Apache 服务器子网
中启动
      "DeviceIndex": "0",
      "GroupSet": [{"Ref": "SecurityGroup"}]
    }]
    "UserData": {"Fn::Base64": {"Fn::Join": ["", [
      "#!/bin/bash -ex\n",
      "yum -y install httpd24-2.4.12\n",      <---从互联网安装Apache
      "service httpd start\n"
    ]]]}
  }
}
```

现在有一个严重的问题：安装Apache不起作用，因为私有子网不能路由到互联网。

## 6.5.5 通过NAT服务器从私有子网访问互联网

公有子网能路由到互联网网关。用户能够使用类似的机制来提供私有子网访问互联网而不需要直接路由到互联网：在公有子网中使用一个NAT服务器，并且创建一条从用户的私有子网到NAT服务器的路由。一台NAT服务器就是一台用来处理网络地址转换的虚拟服务器。来自用户的私有子网的互联网流量将从NAT服务器的公有IP地址访问互联网。

警告

从用户的EC2实例到通过API（Object Store S3，NoSQL数据库DynamoDB）访问的其他AWS服务的流量将通过NAT实例。这很快会成为一个主要的瓶颈。如果用户的EC2实例需要大量与互联网通信，NAT实例可能并不是一个好主意。相反，应该考虑在公有子网中启动这些实例。

为了保持关注点分离，我们将为NAT服务器创建一个新的子网。AWS提供了一个已经为用户配置好了的虚拟服务器映像（AMI）：

```
"SubnetPublicNAT": {
  "Type": "AWS::EC2::Subnet",
  "Properties": {
    "AvailabilityZone": "us-east-1a",
    "CidrBlock": "10.0.0.0/24",      <--10.0.0.0/24 是NAT子网
    "VpcId": {"Ref": "VPC"}
  }
},
"RouteTablePublicNAT": {
  "Type": "AWS::EC2::RouteTable",
  "Properties": {
    "VpcId": {"Ref": "VPC"}
  }
},
[...],
"RoutePublicNATToInternet": {      <--NAT 子网是公有的能路由到互联网
  "Type": "AWS::EC2::Route",
  "Properties": {
    "RouteTableId": {"Ref": "RouteTablePublicNAT"},
    "DestinationCidrBlock": "0.0.0.0/0",
    "GatewayId": {"Ref": "InternetGateway"}
  },
  "DependsOn": "VPCGatewayAttachment"
},
[...],
"NatServer": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    "ImageId": "ami-303b1458",      <--AWS 提供了配置好的NAT 实例映像
    "InstanceType": "t2.micro",
    "KeyName": {"Ref": "KeyName"},
    "NetworkInterfaces": [{
      "AssociatePublicIpAddress": "true",    <--公有IP 地址将会是所有私有子
      "DeleteOnTermination": "true",
      "SubnetId": {"Ref": "SubnetPublicNAT"},
      "DeviceIndex": "0",
      "GroupSet": [{"Ref": "SecurityGroup"}]
    }
  ]
}
```



```

    }},
    "SourceDestCheck": "false"    <--默认情况下，一个实例必须是它发送的网路流
    量的源或目的地。对NAT 实例禁用此检查
  },
  [...],
  "RoutePrivateApacheToInternet": {
    "Type": "AWS::EC2::Route",
    "Properties": {
      "RouteTableId": {"Ref": "RouteTablePrivateApache"},
      "DestinationCidrBlock": "0.0.0.0/0",
      "InstanceId": {"Ref": "NatServer"}    <--从Apache 子网路由到NAT实例
    }
  },
},

```

现在我们已经为使用位于 <https://s3.amazonaws.com/awsinaction/chapter6/vpc.json> 的模板创建 CloudFormation 堆栈做好了准备。一旦完成了这项工作，复制 `VarnishServerPublicName` 输出并在浏览器中打开。我们将看见一个 Varnish 缓存的 Apache 测试页面。

#### 资源清理

在本节结束时别忘了删除堆栈来清除所有用过的资源，否则很可能会因为使用这些资源被收取费用。

## 6.6 小结

- AWS是一个责任共担的环境，在这个环境中只有用户和AWS一起工作才能达到安全。用户负责安全的配置自己的AWS资源以及自己在EC2实例上运行的软件，与此同时AWS保护建筑物和主机系统。
- 使自己的软件保持最新是关键，这样才可以被自动化。
- IAM服务通过AWS API提供身份认证和授权所需要的一切。每一个用户用AWS API发出的请求都通过IAM检查其是否被允许。IAM控制在使用者的AWS账户中谁可以做什么。给自己的用户和角色授予最小的权限可以保护自己的AWS账户。
- 发送到或来自像EC2实例这样的AWS资源的网络流量，在安全组的帮助下可以根据协议、端口以及源和目的地进行筛选。
- 一台堡垒主机是访问用户操作系统的一个入口。它可以用来保护对服务器的SSH访问，可以使用安全组或ACL来实现。
- VPC是AWS里用户拥有完全控制的私有网络。使用VPC，能够控制路由、子网、ACL以及通往互联网的网关或者通过VPN的公司网络。
- 应该在网络中分离关注点来减少潜在的损失。例如，把所有不需要被公有互联网访问的系统放在私有子网中，这样即使用户的某一个子网被攻破了，也可以减少可被攻击的面。

## 第三部分 在云上保存数据

设想一下你的办公室里有个叫独行侠的家伙，他对文件服务器了如指掌。如果独行侠不在办公室，没人能维护文件服务器。当他休假时，如果刚好文件服务器宕机，而领导又需要马上拿到文档，否则公司就会损失一大笔钱的话——麻烦大了，因为没人会知道备份存放在哪。如果独行侠把他的知识存在数据库里，同事们就可以查看到文件服务器的相关信息。但是现在文件如何存储的信息和独行侠紧耦合在一起，就可能无法获取到需要的文件。

现在设想一下服务器上有一些重要的文件存在本地磁盘上。服务器正常运行时一切都好。但是机器不时会出故障——终有一天它们会坏掉。服务器也是如此。如果用户在自己的网站上传一个文件，这文件会存储在哪里？很有可能就保存在服务器的本地磁盘上。但是假设上传到网站的文档，以一个对象的方式持久化保存在独立于服务器的存储上会怎样？如果网站服务器发生故障，用户仍然可以访问到该文档。如果需要两台服务器来承担网站的负载，因为存储没有紧耦合在一台服务器上，所以它们两个都可以访问到文档。只要应用的状态信息存储在服务器以外的地方，用户的系统就具备了容错性和弹性。一些高度专业化的解决方案，如对象存储和数据库，就可以帮助持久化地存储应用状态。

第7章介绍S3，一个对象存储的服务。读者将了解到如何集成对象存储到应用程序中，以实现一个无状态的应用。第8章讨论AWS的虚拟机使用的块存储，以及如何把传统的应用程序部署在块存储上。第9章介绍RDS，一个托管的数据库服务，它支持像Oracle、MySQL、PostgreSQL和Microsoft SQL Server这样的数据库引擎。如果应用需要数据库，使用它可以很轻松地实现无状态的架构。第10章介绍DynamoDB，一个提供NoSQL数据库的服务。用户可以把NoSQL数据库集成到应用里来实现无状态的应用服务器。

## 第7章 存储对象：S3和Glacier

### 本章主要内容

- 使用Terminal终端传输文件到S3
- 使用SDK集成S3到用户的应用程序
- 使用S3服务静态的Web站点
- 研究S3对象存储的内部机制

对象存储可以帮助用户存储图片、视频、文档和可执行文件。本章我们将了解对象存储的基本概念。另外，本章还会介绍AWS提供的对象存储服务Amazon S3，以及备份和归档的存储服务Amazon glacier。

#### 不是所有示例都包含在免费套餐中

本章中的示例不都包含在免费套餐中。当一个示例会产生费用时，会显示一个特殊的警告消息。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

## 7.1 对象存储的概念

以前，数据以文件的形式在层级的目录和文件中进行管理。文件是数据的表现形式。在对象存储里，数据存储为对象。每个对象由一个全球唯一的标识符、元数据和数据本身组成，如图7-1所示。对象的全球唯一标识符也称为键，有了这个全球唯一的标识符，我们才有可能使用分布式系统中的不同设备和机器访问每个对象。

元数据和数据的分离使客户可以直接操作元数据来管理和查询数据。在必要的时候才需要加载数据本身。元数据还用来存储访问权限信息和其他管理任务。

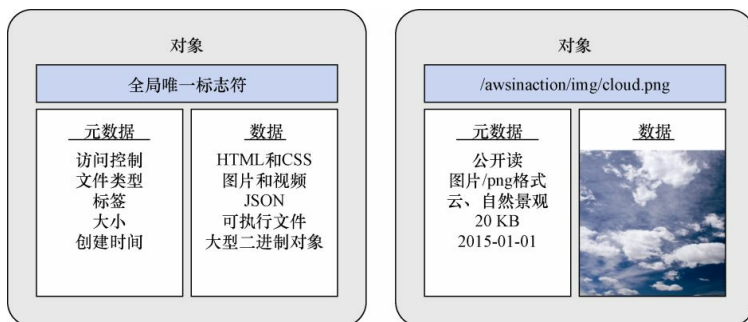


图7-1 对象存储中存放的对象包含3个部分：一个对象ID、描述内容的元数据和内容本身

## 7.2 Amazon S3

Amazon S3是AWS最古老的服务之一。Amazon S3是Amazon Simple Storage Service的简称。它是一个典型的Web服务，让用户可以通过HTTPS和API来存储和访问数据。

这个服务提供了无限存储空间，并且让用户的数据高可用和高度持久化的保存。用户可以保存任何类型的数据，如图片、文档和二进制文件，只要单个对象的容量不超过5TB容量。用户需要为保存在S3的每GB的容量付费，同时还有少量的成本花费在每个数据请求和数据传输流量上。如图7-2所示，可以通过管理控制台使用HTTPS协议访问S3，通过命令行工具（CLI）、SDK和第三方工具来上传和下载对象。



图7-2 通过HTTPS上传和下载对象

S3使用存储桶组织对象。存储桶是对象的容器。用户可以创建最多100个存储桶，每个存储桶都有全球唯一的名字。我指的是真正独一无二的名字——用户必须选择一个没有被其他AWS客户在任何其他区域使用过的存储桶的名字，所以建议选择域名（如`com.mydomain.*`）或者公司名称作为存储桶名的前缀。图7-3解释了这个概念。

典型的使用场景如下。

- 使用S3和AWS CLI来备份和恢复文件。
- 归档对象到Amazon Glacier比归档到S3更节省成本。
- 使用AWS SDK集成Amazon S3到应用程序里，以保存和读取像图片这样的对象。
- 在S3帮助下托管静态网站内容，让所有人都可以访问。

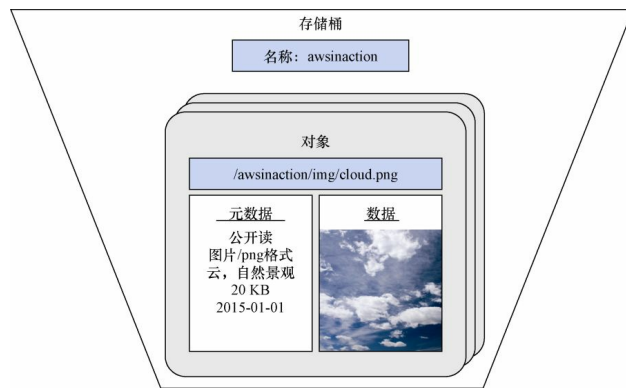


图7-3 S3使用全球唯一命名的存储桶来组织对象

## 7.3 备份用户的数据

关键数据需要及时备份来避免丢失。根据业务要求，用户可能需要备份数据到多个设备或者到另外一个站点。用户可以把任何数据以对象的形式保存在S3，这样就可以把S3当作备份存储使用。

在本节中，读者将了解如何使用AWS CLI命令行工具来上传数据和从S3下载数据。这种方法不仅限于备份的使用场景，其他任何场景都可以使用命令行工具。

首先需要为数据创建一个S3存储桶。像之前提到的那样，存储桶的名字必须避免和其他存储桶冲突，包括其他AWS客户在其他区域的S3存储桶。在终端输入下面的命令行，替换**`$YourName`** 为用户的名字：

```
$ aws s3 mb s3://awsinaction-$YourName
```

所使用的命令应该和下面的命令看上去差不多：

```
$ aws s3 mb s3://awsinaction-awittig
```

如果用户的存储桶名和已有的存储桶冲突，会看到下面的报错信息：

```
A client error (BucketAlreadyExists) [...]
```

在这个例子中，需要使用一个不同的**`$YourName`**。

现在一切就绪，我们可以上传自己的数据了。选择一个我们想要备份的目录，如桌面目录。尽量选择一个合适的目录，其中的文件大小不超过1 GB，并且文件数量少于1000个，这样既不需要等太长时间，也不会超出免费试用的用量。使用下面的命令从本地的目录上传数据到S3存



储桶。将**\$Path** 替换为我们的目录路径，将**\$YourName** 替换为用户的名字。**Sync** 命令比较目录和S3存储桶里的/backup目录，然后只上传新的或者修改过的文件：

```
$ aws s3 sync $Path s3://awsinaction-$YourName/backup
```

所使用的命令应该看上去和下面的类似：

```
$ aws s3 sync /Users/andreas/Desktop s3://awsinaction-awittig/backup
```

根据目录的文件容量和互联网连接的具体情况，上传可能会花费不等的时间。

在文件上传至S3存储桶备份后，可以测试恢复流程。在终端执行下面的命令，将**\$Path** 替换为希望用来恢复的目录（不要使用用来备份的目录），并且将**\$YourName** 替换为自定义的名字。下载目录通常很合适用来测试恢复流程：

```
$ aws s3 cp --recursive s3://awsinaction-$YourName/backup $Path
```

所使用的命令应该和下面的类似：

```
$ aws s3 cp --recursive s3://awsinaction-awittig/backup/ \
/Users/andreas/Downloads/restore
```

同样，根据文件的容量和互联网连接的具体情况，下载可能会花一段时间。

#### 对象的版本

默认情况下，S3存储桶禁用了版本功能。假设使用下面的步骤上传了两个对象。

(1) 添加一个对象，主键A和数据1。

(2) 添加一个对象，主键A和数据2。

如果进行下载，也就是get操作，获取主键为A的对象，将下载到数据2。旧的数据1将不再存在。

读者可以为存储桶激活版本功能来保护数据。下面的命令为存储桶激活版本保护。请替换\$YourName 为自己的名字：

```
$ aws s3api put-bucket-versioning --bucket awsinaction-$YourName \
--versioning-configuration Status=Enabled
```

如果重复之前的步骤，即使在添加了主键A和数据2的对象之后，也可以访问到对象A的第一个版本的数据1。下面的命令获取所有对象和版本：

```
$ aws s3api list-object-versions --bucket awsinaction-$YourName
```

可以下载一个对象的所有版本。

版本在备份和规定的场景中非常有用。记住，需要付费的存储桶的容量将随着新版本的增加而增加。

我们不需要担心丢失数据。S3设计为每年99.999999999%的持久性。

#### 资源清理

读者要执行下面的命令来移除包涵所有备份对象的S3存储桶。读者需要替换\$YourName 为自己的名字来选择正确的存储桶。**rb** 命令移除存储桶，**force** 选项将在删除存储桶之前，强制删除桶里面的每个对象：

```
$ aws s3 rb --force s3://awsinaction-$YourName
```

所使用的命令应该和下面的命令类似：

```
$ aws s3 rb --force s3://awsinaction-awittig
```

完成了！我们已经使用CLI命令行工具上传和下载文件。

#### 移除存储桶造成BucketNotEmpty报错

如果激活了存储桶的版本功能，删除存储桶时将报错BucketNotEmpty。这种情况下请使用管理控制台来删除存储桶。

- (1) 在浏览器中打开管理控制台。
- (2) 从主页面导航至S3服务页面。
- (3) 选择存储桶。
- (4) 从“操作”子菜单中执行“删除存储桶”的操作。

## 7.4 归档对象以优化成本

在前一部分我们使用S3来备份数据。如果希望降低备份存储的成本，应该考虑使用另一个AWS服务Amazon Glacier。在Glacier中存储数据的成本大概是S3中的1/3。但是，它们有哪些区别呢？表7-1展示了S3和Glacier的区别。

表7-1 使用S3和Glacier存储数据的区别

	S3	Glacier
每GB容量成本	0.03美元	0.01美元
数据访问速度	立即可以访问	在提交请求3~5h后
持久性	设计为年度99.999999999%的数据持久性	设计为年度99.999999999%的数据持久性

用户可以通过HTTPS直接使用Glacier服务，或者集成S3一起使用，就像下面的例子展示的一样。

### 7.4.1 创建S3存储桶配合Glacier使用

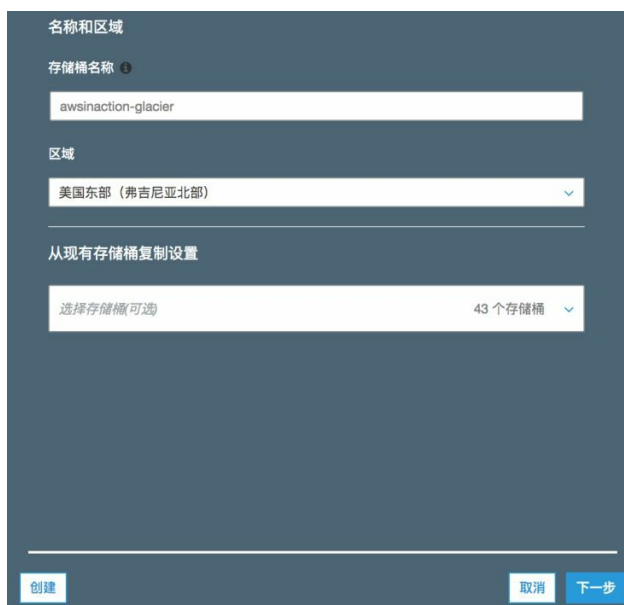
在本部分，读者将了解如何集成S3和Glacier来降低存储数据的成本。如果有异地数据备份的需求，这将很有帮助。首先需要创建一个新的S3存储桶。

- （1）打开管理控制台。
- （2）从主菜单中转移到S3服务页面。

(3) 点击“创建存储桶”按钮。

(4) 为存储桶输入唯一的名字，并选择作为存储桶的区域，如图7-4所示。

(5) 点击“创建”按钮。



The screenshot shows the 'Create Bucket' wizard in the AWS Management Console. The 'Name and Region' section has a text input for the bucket name containing 'awsinaction-glacier' and a dropdown menu for the region set to 'US East (Virginia)'. Below this is a section titled 'Select an existing bucket to copy settings from' with a dropdown menu showing 'Select a bucket to copy' and '43 buckets available'. At the bottom of the wizard are three buttons: 'Create', 'Cancel', and 'Next Step'.

图7-4 使用管理控制台创建S3存储桶

## 7.4.2 添加生命周期规则到存储桶

可以添加一条或者多条生命周期规则到存储桶，以管理对象的生命周期。生命周期规则可以用来在给定的日期之后归档或者删除对象，它还可以帮助把S3的对象归档到Glacier。

添加一条生命周期规则来移动对象到Glacier，参照下面的具体步骤。

(1) 在管理控制台打开S3服务页面。

(2) 点击进入创建的存储桶，并选择“管理”。

(3) 在“管理”标签栏的下方，点击“添加生命周期规则”按钮，如

图7-5所示。

**警告**

Glacier不包含在免费试用内。本示例将造成一些成本花费。如果想了解更多现在的定价信息，可访问AWS官方网站。



图7-5 添加生命周期管理规则来自动移动对象到Glacier

将弹出一个向导帮助为存储桶创建新的生命周期管理规则。第一步是选择生命周期规则的目标，输入规则名称为**move-to-glacier**，在筛选条件文本框保持空白，以将生命规则应用到整个存储桶，如图7-6所示，并点击“下一步”。

图7-6 选择生命周期规则的目标

下一步是配置生命周期规则。选择“当前版本”为配置转换的目标，并点击“添加转换”，接着选择“转换到Glacier前经过.....”。为了尽快触发生命周期规则来让对象一旦创建就归档，选择在对象创建0天后进行转换，如图7-7所示。连续点击“下一步”进入向导的最后一步。

图7-7 选择生命周期规则的时间

另外，如图7-8所示检查规则的细节。如果一切都没有问题，点击“保存”按钮。

The screenshot shows the 'Lifecycle Rules' (生命周期规则) configuration page in the AWS console. The page has a blue header with a close button (X) and four tabs: '名称和范围' (Name and Scope), '转换' (Transition), '过期' (Expiration), and '审核' (Review), with the '审核' tab selected. The main content area is divided into three sections: '名称和范围' (Name and Scope) with fields for '名称' (Name) set to 'move-to-glacier' and '范围' (Scope) set to '整个存储桶' (Entire bucket), both with '编辑' (Edit) links; '转换' (Transition) with the text '对于对象的当前版本' (For the current version of the object) and '转换到 Amazon Glacier 经过的时间 0 天' (Transition to Amazon Glacier after 0 days), also with an '编辑' (Edit) link; and '过期' (Expiration) with an '编辑' (Edit) link. Below these sections is a summary box stating '检查规则 move-to-glacier。它包含重叠的前缀。' (Check rule move-to-glacier. It contains overlapping prefixes.). At the bottom right are '上一步' (Previous step) and '保存' (Save) buttons.

图7-8 保存S3存储桶的生命周期规则

### 7.4.3 测试Glacier和生命周期规则

我们已经成功地创建了生命周期管理规则，它将自动把对象从S3存储桶移动至Glacier。

#### 注意

移动对象到Glacier大概会花费24h左右的时间。从Glacier恢复数据到S3大概需要3~5h，所以读者尽可以继续阅读本书，而无须执行上面的示例。

打开存储桶，在管理控制台点击“上传”来上传文件到存储桶。在图7-9中，我们已经上传了3个文件到S3。默认情况下，所有文件都保存在“标准”存储类别，意味着它们目前保存在S3中。





图7-9 生命周期规则将在几小时后移动对象到Glacier

生命周期规则将移动对象到Glacier。但是，即使把时间设为0天，移动过程仍然会需要24h左右。在对象移动到Glacier之后，存储类别会切换为Glacier。

用户无法直接下载存储在Glacier中的文件，但是可以触发一个恢复过程来从Glacier恢复对象到S3。参考图7-10所示的步骤在管理控制台触发恢复操作。

- （1）打开S3存储桶。
- （2）选择希望从Glacier恢复的对象并点击“更多”按钮。
- （3）选择“启动还原”。
- （4）在弹出的对话框里选择对象从Glacier恢复后要保留在S3的天数，选择要检索的速度选项（标准检索3~5h），如图7-10所示。
- （5）点击“还原”来发起恢复。

恢复对象大概需要3~5h。在恢复完成后，可以下载对象。



图7-10 从Glacier里恢复对象到S3非常简单，单默认获取选项需要3~5h

### 资源清理

完成Glacier示例之后用户需要删除自己的存储桶。从管理控制台按照如下操作可以完成删除。

- (1) 在管理控制台打开S3服务。
- (2) 点击选择已经创建的存储桶。
- (3) 然后点击“删除存储桶”按钮。
- (4) 在文本框内输入存储桶名称，并点击“确认删除”。

我们已经了解了如何使用CLI和管理控制台来使用S3。接下来我们来演示一下如何通过SDK集成S3到自己的应用程序。

## 7.5 程序的方式存储对象

S3可以通过HTTPS和API来访问。这意味着，用户可以集成S3到应用程序里，用程序调用API来提交请求到S3。如果用户使用的是一个常见的编程语言，如Java、JavaScript、PHP、Python、Ruby或者.Net，就可以免费使用AWS提供的SDK。在应用程序里通过SDK的帮助可以完成下面的操作。

- 列出存储桶和里面的对象。
- 创建、更新和删除（CRUD）对象和存储桶。
- 管理对象的访问权限和生命周期。

用户可以在下面的场景中集成S3到应用程序。

- 允许用户上传一个档案图片。在S3上保存图片，并让它可以公开访问。通过HTTPS集成图片到自己的网站。
- 生成月度报表（如PDF文件）并把它们开放给用户访问。创建文档并上传到S3。如果用户想要下载文档，从S3获取这些文件。
- 在不同的应用之间共享数据。用户可以从不同的应用中访问文档。例如，应用A写入最新的销售信息到文档里，应用B可以下载该文档并分析数据。

集成S3到应用程序可以帮助实现无状态的服务器的概念。在本节中，我们将深入了解一个简单的名字叫Simple S3 Gallery的互联网应用。这个互联网应用搭建在Node.js并且使用面向JavaScript和Node.js的AWS SDK。因为概念相似，读者可以轻松地把本部分学到的东西转移到其他编程语言。图7-11展示了Simple S3 Gallery的图形界面。我们现在设置S3来开始搭建。

## Simple S3 Gallery

### Upload

No file selected.

### Images



图7-11 Simple S3 Gallery的应用允许用户上传图片到S3存储桶，然后从存储桶下载来展示图片

## 7.5.1 设置S3存储桶

开始，用户需要创建一个空的存储桶。执行下面的命令，替换`$YourName` 为用户的名字或者昵称：

```
$ aws s3 mb s3://awsinaction-sdk-$YourName
```

存储桶已经准备好了。下一步安装互联网应用。

## 7.5.2 安装使用S3的互联网应用

读者可以在本书的代码目录里的`/chapter7/gallery/` 找到Simple S3 Gallery的代码。切换到该目录，在终端运行`npm install` 安装所有的依赖包。

要运行Web应用，需要运行下面的命令。将`$YourName` 替换为用户的名字，将S3的存储桶名传递给Web应用程序：

```
$ node server.js awsinaction-sdk-$YourName
```

启动服务器后，使用浏览器访问<http://localhost:8080>打开Gallery应用。试着上传一张新图片。

### 7.5.3 检查使用SDK访问S3的代码

我们已经看到Simple S3 Gallery如何上传和显示在S3的图片。查看一下部分代码将有助于了解怎样才能集成S3到应用程序。如果无法完全理解编程语言（JavaScript）和Node.js平台的实现细节，这没有关系，只要大概了解如何通过SDK使用S3即可。

#### 1. 上传一个图片到S3

用户可以调用S3服务SDK中的`putObject()`方法来上传一个图片。应用程序将连接到S3服务并且使用HTTPS协议来传输图片。代码清单7-1列出如何完成这些操作。

代码清单7-1 使用AWS SDK上传图片到S3

```
[...]
var AWS = require("aws-sdk");      <--插入AWS SDK
[...]
var s3 = new AWS.S3({"region": "us-east-1"});    <--配置AWS SDK

var bucket = "[...]";

function uploadImage(image, response) {
  var params = {      <--上传图片的参数
    Body: image,      <--图片内容

    Bucket: bucket,    <--存储桶的名称
    Key: uuid.v4(),    <--允许所有人从存储桶读取图片
    ACL: "public-read", <--为对象生成一个唯一的主键
    ContentLength: image.byteCount    <--图片的大小，以字节为单位
  };
  s3.putObject(params, function(err, data) {    <--上传图片到S3
```

```

    if (err) {      <--处理错误（如网络问题）
      console.error(err);
      response.status(500);
      response.send("Internal server error.");
    } else {      <--操作成功的后续操作
      response.redirect("/");
    }
  });
}
[...]
```

AWS SDK负责在后台发送所有相关的HTTPS请求到S3 API。

## 2. 列出S3存储桶的所有图片

为了列出所有图片，应用程序需要列出用户的存储桶的所有对象。这可以通过调用S3服务的`listObjects()`方法完成。代码清单7-2显示`server.js`的JavaScript代码的相应部分实现，这是Web服务器端的代码。

代码清单7-2 获取S3存储桶的所有图片地址

```

[...]
```

```

var bucket = "[...]";

function listImages(response) {
  var params = {      <--定义list-objects 方法的参数
    Bucket: bucket
  };
  s3.listObjects(params, function(err, data) {      <--调用list-objects 方法
    if (err) {
      console.error(err);
      response.status(500);
      response.send("Internal server error.");
    } else {
      var stream = mu.compileAndRender("index.html",
        {
          Objects: data.Contents,      <--返回结果包含存储桶的对象列表
          Bucket: bucket
        }
      );
      stream.pipe(response);
    }
  }
}
```

```
});  
}
```

1 `list-objects`操作返回存储桶的所有图片，但是该代码清单不包括图片的内容。在上传阶段，图片的访问权限设置为**Public Read**公开访问。这意味着任何人都可以通过存储桶的名字和对象的主键直接从S3下载图片。代码清单7-3展示了`index.html`模板的部分代码，这段代码将渲染该请求。`Objects` 变量包含了存储桶的所有对象。

代码清单7-3 把数据渲染成HTML的模板

```
[...]
<h2>Images</h2>
{{#Objects}}      <-- 遍历所有对象
  <p>              <-- 构建URL，用来从存储桶中提取一个图片
    
  </p>
{{/Objects}}
[...]
```

现在我们可以看到Simple S3 Gallery应用里和S3集成的3个重要的部分：上传一个图片、列出所有的图片和下载一个图片。

#### 资源清理

别忘了清理和删除在本例中的S3存储桶。使用下面的命令，将`$YourName` 替换为自己的名字：

```
$ aws s3 rb --force s3://awsinaction-sdk-$YourName
```

我们已经学会了如何在AWS SDK的帮助下将S3用于Java Script和Node.js。针对其他编程语言使用AWS SDK也是类似的。



## 7.6 使用S3来实现静态网站托管

可以使用S3来服务一个静态的网站，并且服务静态内容，如HTML、CSS、图片（如PNG和JPG）、音频和视频。不能在S3上执行服务器端脚本（如PHP或者JSP），但是可以在客户端执行存储在S3上的客户端脚本（如JavaScript）。

### 通过使用CDN内容分发系统来改善速度

使用内容分发系统帮助减少静态网站内容的加载时间。CDN在全球范围分发HTML、CSS和图片这样的静态内容。一旦用户请求访问静态内容，CDN可以从最近的位置以最低的时延返回结果给用户。

Amazon S3不是一个CDN，但是可以让S3作为AWS的CDN服务：Amazon CloudFront的源服务器。如果读者希望了解如何设置CloudFront，可以从AWS官方网站查看CloudFront的文档，本书不会介绍这部分内容。

另外，S3还提供了一些服务静态网站的功能。

- 指定自定义的index文档和error文档。
- 定义对所有或者特定的页面请求进行重定向。
- 为S3存储桶设置自定义的域名。

### 7.6.1 创建存储桶并上传一个静态网站

首先需要创建一个新的S3存储桶。打开终端并且执行下面的命令来完成这个步骤。替换**\$BucketName** 为自己的存储桶名称。如之前提到的，存储桶的名字需要在全球唯一，所以一个明智的做法是用自己的域名作为存储桶的名字（如static.yourdomain.com）。如果希望重定向域名到S3，必须使用域名作为S3的存储桶名：

```
$ aws s3 mb s3://$BucketName
```

现在存储桶是空的，接下来将存一个HTML文档进去。我们已经准备好一个HTML文件（`helloworld.html`）。读者可以下载的源代码中找到该文件。

现在读者可以上传文件到S3。读者可以执行下面的命令来做到这一点，将`$PathToPlaceholder` 替换为之前下载HTML的路径并且替换`$BucketName` 为用户的存储桶名：

```
$ aws s3 cp $PathToPlaceholder/helloworld.html \
s3://$BucketName/helloworld.html
```

现在已经创建了一个存储桶并且上传了`helloworld.html`的HTML文档。接下来配置存储桶。

## 7.6.2 配置存储桶来实现静态网站托管

默认情况下，只有文件的拥有者可以访问S3存储桶的文件。使用S3来提供静态网站服务的话，就需要允许所有人查看或者下载该存储桶里的文档。存储桶策略 可以用来在全局控制存储桶里对象的访问权限。在第6章里我们已经学习了IAM的策略：IAM策略使用JSON定义权限，它包含了一个或者多个声明，并且一个声明里允许或者拒绝特定操作对某个资源的访问。存储桶策略和IAM策略很相似。

读者可以下载的源代码中找到存储桶策略`bucketpolicy.json`。

接下来需要编辑`bucketpolicy.json`文件。代码清单7-4解释了该策略。使用自己偏好的编辑器打开该文件，并且替换`$BucketName` 为具体的存储桶名。

代码清单7-4 本存储桶策略允许对存储桶里的所有对象的只读访问

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AddPerm",
```

```
"Effect": "Allow",      <-- 允许访问.....
"Principal": "*",      <-- .....任何人.....
"Action": ["s3:GetObject"],    <-- .....去下载对象.....
"Resource": ["arn:aws:s3:::$BucketName/ *"]    <-- .....从你的存储桶中

}
]
}
```

使用下面的命令可以添加桶策略到存储桶。将**\$BucketName** 替换为存储桶名，并且将**\$PathToPolicy** 替换为bucketpolicy.json的路径：

```
$ aws s3api put-bucket-policy --bucket $BucketName \
--policy file://$PathToPolicy/bucketpolicy.json
```

现在存储桶里的所有对象可以被任何人下载。接下来需要激活和配置S3服务静态网站。要做到这一点，需要执行下面的命令，并且替换**\$BucketName** 为实际的存储桶名称：

```
$ aws s3 website s3://$BucketName --index-document helloworld.html
```

存储桶现在已经配置为服务一个静态网站，使用helloworld.html作为索引页面。下面来了解如何访问该网站。

### 7.6.3 访问S3上托管的静态网站

可以通过浏览器访问静态网站。要先选择正确的端点。根据存储桶所在区域的不同，S3静态网站的端点也可能不同：

```
$BucketName.s3-website-$Region.amazonaws.com
```

这个存储桶创建在默认的区域us-east-1，所以输入**\$BucketName** 来

组成存储桶的端点，替换\$Region 为us-east-1：

```
$BucketName.s3-website-us-east-1.amazonaws.com
```

使用浏览器打开这个URL，应该能看的一个Hello World的网站。

#### 关联一个自定义的域名到S3的存储桶

如果不想使用awsinaction.s3-website-us-east-1.amazonaws.com这样的域名作为静态网站的域名，用户可以关联一个自定义的域名到S3存储桶。用户只需要为自己的域名添加一个CNAME别名记录，让该记录指向S3存储桶的端点即可。

CNAME别名记录只在满足下面条件的时候生效。

- 存储桶名必须和CNAME别名记录一样。例如，要创建一个CNAME给static.yourdomain.com，存储桶名也必须是static.yourdomain.com。
- CNAME别名记录不适用于主域名。可以给予域名创建别名记录的资源，如static或者www这样前缀的域名。如果想关联主域名到S3存储桶，需要使用AWS提供的Route 53的DNS服务。

#### 资源清理

别忘了在完成本示例的时候清理所用的资源。要做到这一点，执行下面的命令，将\$BucketName 替换为自己的存储桶名：

```
$ aws s3 rb --force s3://$BucketName
```

## 7.7 对象存储的内部机制

在使用CLI命令行访问S3的时候，了解S3对象存储的一些内部机制会有帮助。和其他很多对象存储不同的是，S3是最终一致的。如果不考虑到这一点，在刚刚更新对象之后马上去访问对象，读者会观察到奇怪的结果。另外一个挑战是如何合理地设计对象主键来实现S3的I/O性能的最大化。接下来我们就来了解一下这两点。

### 7.7.1 确保数据一致性

S3上创建、更新或者删除对象的操作是原子操作。这意味着，如果用户在创建、更新或者删除之后读取这个对象，永远不会读到失效的或者一半的数据。但是有可能读取操作会在一段时间里只返回旧的数据。

S3提供的是最终一致性。如果上传一个已有对象的新的版本，并且S3对该请求返回成功代码，意味着数据已经安全的保存在S3。但是，如图7-12所示，立即去下载更新后的对象仍可能返回旧的版本。如果反复重试下载对象的操作，过段时间就可以下载到更新后的对象。

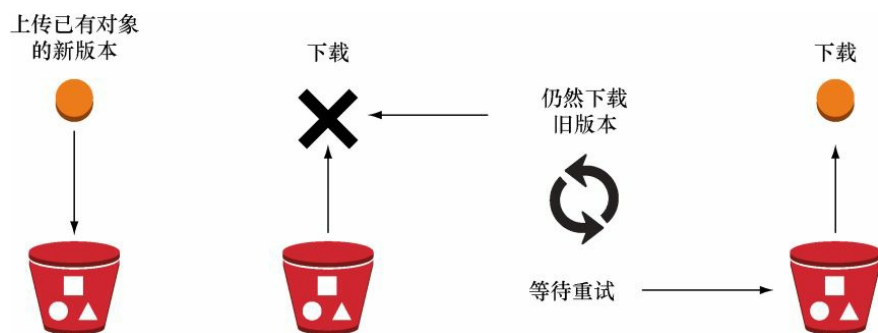


图7-12 最终一致性：如果更新一个对象然后尝试读取，对象可能还包含旧的版本。在有些情况下，最新的版本可以访问到

在上传新对象之后，立即提交的读请求会读到一致的数据。但是在更新或者删除操作之后的读请求操作将返回最终一致的结果。

## 7.7.2 选择合适的键

给变量或者文件取名是IT领域最困难的任务之一。为存储在S3上的对象选择合适的键尤其困难。键的命名决定了该键保存在哪一个分区。为所有对象的键在开头的部分使用相同的字符串，将限制S3存储桶的最大I/O性能。相反，应该为对象选择开头不同的字符串作为键。如图7-13所示，这会带来最大的I/O性能。

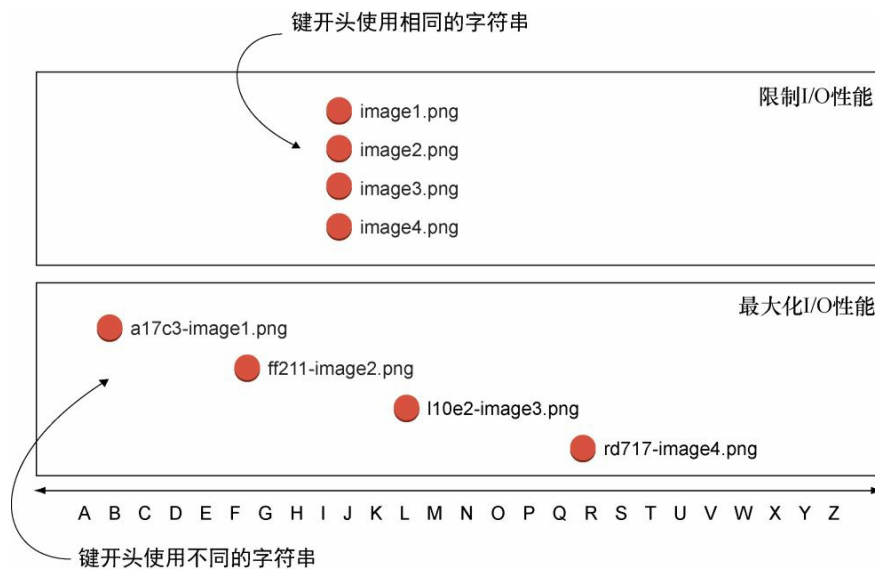


图7-13 为了改善S3的I/O性能，不要使用开头相同的字符串作为键

在键中使用一个斜线 (/) 的效果就像为对象创建目录一样。如果用户创建的对象键为 `folder/object.png`，在通过管理控制台这样的图形化界面浏览存储桶的时候，用户会看到目录。但是从技术的本质上看，对象的键仍然是 `folder/object.png`。

假设要存储的图片分别由不同的用户上传，我们为对象的键设计了下面的命名方式：

```
$ImageId.png
```

`$ImageId` 是一个递增的字符ID。对象列表看上去大概如下：

```
/image1.png  
/image2.png  
/image3.png  
/image4.png
```

对象的键为阿拉伯字母顺序排序，这种情况下S3的存储桶的最大性能不会达到最优。可以通过为每个对象的键添加散列前缀的方式修复这个问题。例如，可以使用原始键名的MD5的散列附加在后面组成新的键：

```
/a17c3-image1.png  
/ff211-image2.png  
/110e2-image3.png  
/rd717-image4.png
```

这样会有助于分配对象键到不同的分区，从而提高S3的I/O性能。了解这些S3的内部机制有助于优化对其的使用。

## 7.8 小结

- 对象由唯一的标识符、用来描述和管理对象的元数据和内容本身组成。图片、文档、可执行文件或者任何其他内容都可以用对象的形式保存在对象存储中。
- Amazon S3是一个对象存储，通过HTTPS访问。可以使用CLI命令行工具、SDK开发者工具包或者管理控制台来上传、管理和下载对象。
- 因为不再需要把对象保存在本地服务器中，所以在应用程序中集成S3有助于实现无状态的服务器架构。
- 可以定义一个生命周期管理的规则，把数据自动从Amazon S3移动到Amazon Glacier，从而降低数据存储的成本。Amazon Glacier是一个很特别的，适合存放不经常访问的归档数据的服务。
- S3是最终一致的存储。在应用程序中集成S3的时候应该考虑到这一点，并且相应地处理以避免出现不希望的结果。



## 第8章 在硬盘上存储数据：EBS和实例存储

### 本章主要内容

- 附加网络存储到EC2实例
- 使用EC2实例的实例存储
- 备份块级别存储
- 测试和调试块级别存储的性能
- 比较实例存储和网络附加存储

用户就像在个人电脑上做的那样，可以使用磁盘文件系统（FAT32、NTFS、ext3、ext4、XFS等）和块级别存储来存储文件。块是顺序的字节和最小的寻址单位。操作系统位于需要访问文件的应用程序和底层的文件系统和块存储的中间。文件系统负责管理文件放在底层的块级别存储的具体哪个位置（哪个块的地址）。块级别的存储只能在运行操作系统的EC2实例上使用。

操作系统通过打开、写和读系统调用来提供对块级别存储的访问。简化后的读请求操作大概是下面这样的。

（1）应用程序想要读取文件 `/path/to/file.txt`，然后提交了一个读系统调用。

（2）操作系统转发读请求给文件系统。

（3）文件系统把 `/path/to/file.txt` 文件翻译为具体存储数据的磁盘的数据块。

数据库这样的应用程序通过使用系统调用的方式来读写文件，它们必须能够访问块级别的存储来持久化保存数据。因为MySQL必须使用系统调用访问文件，所以不能把MySQL的数据库的文件保存在对象存储里。

#### 不是所有示例都包含在免费套餐中

本章中的示例不都包含在免费套餐中。当一个示例会产生费用时，会显示一个特殊的警告消息。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

AWS提供两种类型的块级别存储，即网络附加存储和实例存储。网络附加存储（就像iSCSI）通过网卡附加到EC2实例，但是实例存储是提供你的EC2实例的主机系统提供的正常的物理磁盘。大多数情况下，网络附加存储是最好的选择，因为它为数据提供99.999%的可用性。实例存储在需要性能的时候会更合适。8.1节至8.3节会介绍和比较两种块级别存储解决方案。我们将块级别存储连接到EC2实例，进行性能测试，并探讨如何备份数据。之后，你将使用实例存储和网络附加存储来搭建共享的文件系统。

## 8.1 网络附加存储

弹性数据块存储（EBS）提供网络附加的，数据块级别的存储，并且提供99.999%的可用性。图8-1展示了如何在EC2实例上使用EBS卷。



图8-1 EBS卷是独立的资源，但是可以挂载到一个EC2实例上使用

EBS卷：

- 不属于EC2实例的一部分，它们通过网卡附加到EC2实例。如果终结了EC2实例，EBS卷仍然存在；
- 可以独立存在或者同一时间挂载到一个EC2实例上；
- 可以像普通硬盘一样使用；
- 类似于RAID1，在后台把数据保存到多块磁盘上。

### 警告

不能同时挂在一块EBS卷到多台服务器！

### 8.1.1 创建EBS卷并挂载到服务器

下面的示例演示了如何在CloudFormation的帮助下创建EBS卷，并且挂载到EC2实例。

```
"Server": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    [...]
  }
},
"Volume": {
```

```

    "Type": "AWS::EC2::Volume",          <---EBS 卷描述
    "Properties": {
      "AvailabilityZone": {"Fn::GetAtt": ["Server", "AvailabilityZone"]},
      "Size": "5",              <---5 GB 容量
      "VolumeType": "gp2"      <---基于SSD
    }
  },
  "VolumeAttachment": {
    "Type": "AWS::EC2::VolumeAttachment",    <---附加EBS 卷到服务器
    "Properties": {
      "Device": "/dev/xvdf",      <---设备名
      "InstanceId": {"Ref": "Server"},
      "VolumeId": {"Ref": "Volume"}
    }
  }
}

```

EBS卷是一个独立的资源。这意味它可以独立于EC2服务器存在，但是需要一台EC2服务器才能使用EBS卷。

## 8.1.2 使用弹性数据块存储

为了帮助读者了解EBS，我们准备了一个CloudFormation的模板，其位于<https://s3.amazonaws.com/awsinaction/chapter8/ebs.json>。基于这个模板创建一个堆栈，设置**AttachVolume** 参数为**yes**，然后复制**Public-Name** 输出，并且通过SSH进行连接。

使用**fdisk** 可以看到已经附加的EBS卷。通常，EBS卷可以在/dev/xvdf到/dev/xvdp下面找到。根卷（/dev/xvda）是一个例外——在启动EC2实例的时候，它基于选择的AMI创建，并且包含了所有用于引导实例的信息（操作系统文件）：

```

$ sudo fdisk -l
Disk /dev/xvda: 8589 MB [...]    <---根卷（存放操作系统）
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: gpt

#      Start      End      Size   Type        Name

```

```
1      4096    16777182      8G    Linux filesystem Linux
128     2048      4095      1M    BIOS boot parti BIOS Boot Partition
```

```
Disk /dev/xvdf: 5368 MB [...]    <--附加的EBS 卷
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

创建一个新的EBS卷时，必须在上创建一个文件系统。你还可以在EBS卷上创建不同的分区，但是在本例中卷的容量只有5 GB，所以不需要进一步分区。分区不是使用EBS卷的最佳实践。应该创建和需求相同容量大小的卷；在需要两个单独的分区情况下，直接创建两个卷更合适。在Linux中，可以使用mkfs 基于卷来创建文件系统。下面的例子创建了ext4的文件系统：

```
$ sudo mkfs -t ext4 /dev/xvdf
mke2fs 1.42.12 (29-Aug-2014)
Creating filesystem with 1310720 4k blocks and 327680 inodes
Filesystem UUID: e9c74e8b-6e10-4243-9756-047ceaf22abc
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912, 819200, 884736

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

文件系统创建完成之后，就可以挂载文件系统到一个目录：

```
$ sudo mkdir /mnt/volume/
$ sudo mount /dev/xvdf /mnt/volume/
```

使用df -h 命令可以查看已经挂载的卷：

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/xvda1      7.8G  1.1G  6.6G  14% /    <--根卷（存放操作系统）
```

```
devtmpfs    490M    60K    490M    1% /dev
tmpfs       499M      0    499M    0% /dev/shm
/dev/xvdf   4.8G    10M    4.6G    1% /mnt/volume    <--EBS 卷
```

EBS卷有一个很大的优势：它们不属于EC2的一部分，是独立的资源。我们可以保存文件到EBS卷，然后去掉挂载，并从EC2上摘掉该卷，这样就可以了解到它独立于EC2的特性：

```
$ sudo touch /mnt/volume/testfile    <--在/mnt/volume/目录中创建testfile
$ sudo umount /mnt/volume/
```

现在更新CloudFormation堆栈，修改AttachVolume 参数为no 。这个操作将从EC2上摘掉EBS卷。在堆栈更新完成后，EC2上只剩下系统根卷：

```
$ sudo fdisk -l
Disk /dev/xvda: 8589 MB, 8589934592 bytes, 16777216 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: gpt

#           Start          End      Size Type              Name
  1         4096       16777182      8G Linux filesystem Linux
128         2048          4095      1M BIOS boot parti BIOS Boot Partition
```

/mnt/volume/中的测试文件也不见了：

```
$ ls /mnt/volume/testfile
ls: cannot access /mnt/volume/testfile: No such file or directory
```

现在可以重新挂在EBS卷到EC2。更改CloudFormation堆栈，并修改AttachVolume 参数为Yes 。在更新完成后，/dev/xvdf重新可以访问：

```
$ sudo mount /dev/xvdf /mnt/volume/
$ ls /mnt/volume/testfile      <---检查/mnt/volume/目录中是否有testfile
/mnt/volume/testfile
```

太棒了。在/mnt/volume/下面创建的测试文件还在那里。

### 8.1.3 玩转性能

硬盘的性能测试通常分为读操作和写操作测试。用户可以使用很多不同的工具进行测试。一个简单的工具是**dd**，它可以通过指定数据源**if=/**到源路径和目标**of=/**到目的路径来进行数据块级别的读写测试。

```
$ sudo dd if=/dev/zero of=/mnt/volume/tempfile bs=1M count=1024 \      <---
每次写1 MB，进行1 024 次写测试
  conv=fdatasync,notrunc
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 16.9858 s, 63.2 MB/s      <---63.2 MB/s 的
写性能

$ echo 3 | sudo tee /proc/sys/vm/drop_caches      <---缓存清空至磁盘

$ sudo dd if=/mnt/volume/tempfile of=/dev/null bs=1M count=1024      <---每
次读1 MB，进行1 024 次读测试
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 16.3157 s, 65.8 MB/s      <---65.8 MB/s 的
读性能
```

注意，随着真实的工作负载的不同，存储性能的表现也不一样。本示例假设文件大小为**1MB**。如果应用是互联网网站的话，很有可能处理数据的会是大量的小文件。

但是EBS卷更加复杂。存储性能取决于EC2的实例类型和EBS卷的类型。表8-1列出了默认为EBS优化的EC2实例类型和EBS卷的类型，有些EC2实例类型可能需要每小时为EBS优化付出额外的成本。每秒的I/O

操作使用16 KB的I/O大小来进行测量。存储性能很大程度上依赖于实际的工作负载：是读操作还是写操作，每个I/O操作的大小。这些数据仅供参考，生产中的性能情况可能有所差异。

表8-1 EBS优化的实例类型的性能表现

使用场景	实例类型	最大带宽 (MB/s)	每秒最大I/O次数	默认EBS优化
通用类型	m3.xlarge~c4.large	60~120	4 000~8 000	否
优化的计算	c3.xlarge~3.4xlarge	60~240	4 000~16 000	否
优化的计算	c4.large~c4.8xlarge	60~480	4 000~32 000	是
优化的内存	r3.xlarge~r3.4xlarge	60~240	4 000~16 000	否
优化的存储	i2.xlarge~i2.4xlarge 60	60~240	4 000~16 000	否
优化的存储	d2.xlarge~d2.8xlarge	90~480	6 000~32 000	是

根据工作负载的要求，你需要选择一个能够提供足够带宽的EC2实例类型。另外，EBS卷必须能够充分使用EC2提供的带宽。表8-2给出了可以选择的EBS卷类型和它们的性能指标。

表8-2 不同的EBS类型

EBS卷类型	大小	最大吞吐量 (MiB/s)	IOPS	突发IOPS性能	价格



物理磁盘	1 GiB~1 TiB	40~90	100	几百	\$
通用类型 (SSD)	1 GiB~16 TiB	160	3/GiB (最高10 000)	3 000	\$\$
预配置IOPS性能 (SSD)	4 GiB~16 TiB	320	同预配置 (最高30/GiB即20 000)	—	\$\$\$

不管实际使用了多少容量，EBS卷都按照卷的容量大小来收费。如果创建了一个100 GB大小的EBS卷，即使没有保存任何数据在上面，你仍然需要为100 GB的EBS卷进行付费。如果使用的是物理磁盘，则需要为每次I/O操作付费。如果使用的预配置IOPS性能的EBS卷（SSD磁盘），还需要为预先配置的IOPS性能付费。用户可以使用简单月度成本计算器来计算存储成本。

#### GiB和TiB

GiB和TiB的术语并不经常看到；你可能更加熟悉GB和TB的术语。但是在某些情况下，AWS使用GiB和TiB的术语。下面是这两个术语的含义：

- 1 GiB =  $2^{30}$  字节 = 1 073 741 824字节
- 1 GiB约为1.074 GB
- 1 GB =  $10^9$  字节 = 1 000 000 000字节

我们建议默认使用通用SSD磁盘。如果工作负载需要更多的IOPS性能，推荐选择预配置IOPS性能的SSD磁盘。用户可以附加多个EBS卷到一台EC2实例来增加容量或者总体的性能。

可以把两个甚至更多的EBS卷挂载到同一台EC2，并使用软件RAID0来提升性能。RAID0技术让数据分散到多块磁盘，但是同一个数据仅存储在一块磁盘上。在Linux系统中可以使用mdadm 来创建软件RAID。

## 8.1.4 备份数据

EBS卷提供99.999%的可用性，但是仍然需要不时地创建备份。幸运的是，EBS卷提供了优化的易于使用的EBS快照功能来备份EBS卷的数据。快照是存储在S3上的块级别的数据复制。如果卷大小是5 GB并且上面保存了1 GB的数据，第一个快照的容量大小为1 GB左右。在创建第一个快照后，只有更改过的数据才会被保存在S3，以节省备份的容量。EBS卷的快照收费取决于你使用的GB容量。

可以使用下面的命令行来创建快照。在创建快照之前，需要获取EBS卷的ID。可以在CloudFormation的输出内容里找到VolumeId 卷ID，或者运行下面的命令：

```
$ aws --region us-east-1 ec2 describe-volumes \
--filters "Name=size,Values=5" --query "Volumes[].VolumeId" \
--output text
vol-fd3c0aba      <---用户的$VolumeId
```

有了卷ID，可以接下来创建一个快照：

```
$ aws --region us-east-1 ec2 create-snapshot --volume-id $VolumeId      <---
替换为用户的$SnapshotId
{
  "Description": null,
  "Encrypted": false,
  "VolumeId": "vol-fd3c0aba",
  "State": "pending",          <---用户的快照的状态
  "VolumeSize": 5,
  "Progress": null,
  "StartTime": "2015-05-04T08:28:18.000Z",
  "SnapshotId": "snap-cde01a8c",    <---用户的$SnapshotId
  "OwnerId": "878533158213"
}
```

根据卷的容量大小和改变的数据量的不同，创建快照的时间也不一样。我们可以使用下面的命令来查看快照的状态：

```
$ aws --region us-east-1 ec2 describe-snapshots --snapshot-ids $SnapshotId
```

```

    <-- 替换为用户的$SnapshotId
{
  "Snapshots": [
    {
      "Description": null,
      "Encrypted": false,
      "VolumeId": "vol-fd3c0aba",
      "State": "completed",      <-- “completed”代表快照创建完成
      "VolumeSize": 5,
      "Progress": "100%",      <-- 用户的快照的进度
      "StartTime": "2015-05-04T08:28:18.000Z",
      "SnapshotId": "snap-cde01a8c",
      "OwnerId": "878533158213"
    }
  ]
}

```

可以在一个已经挂载并且正在使用的EBS卷上创建快照，但是如果内存缓存中还有尚未写入磁盘的数据，这可能带来一些问题。如果必须在EBS卷使用的时候创建快照，可以使用下面的步骤来安全地创建快照。

(1) 在服务器上运行 **fsfreeze -f /mnt/volume** 命令来冻结所有写操作。

(2) 创建快照。

(3) 使用 **fsfreeze -u /mnt/volume** 命令来恢复写操作。

(4) 等待快照操作完成。

用户只需要在开始请求创建快照的时候冻结I/O。从一个AMI（AMI是一个快照）创建EC2的时候，AWS使用EBS快照来创建一个新的EBS卷（根卷）。

要恢复快照里的数据，你必须基于快照来创建一个新的EBS卷。当用户从AMI来创建一台EC2实例时，AWS基于快照来创建一个新的EBS卷（AMI是一个快照）

资源清理

别忘了删除快照：

```
$ aws --region us-east-1 ec2 delete-snapshot --snapshot-id $SnapshotId
```

在完成这一部分的时候还需要删除整个堆栈以清理所有用过的资源；否则将会为用到的资源付费。

## 8.2 实例存储

实例存储像物理磁盘一样提供块级别的存储。像图8-2显示的那样，实例存储是EC2的一部分，并且只有在EC2正常运行时才可用。如果停止或者终结实例，上面的数据不会持久化保存，所以不需要为实例存储单独付费，实例存储的价格包含在EC2实例的价格里。

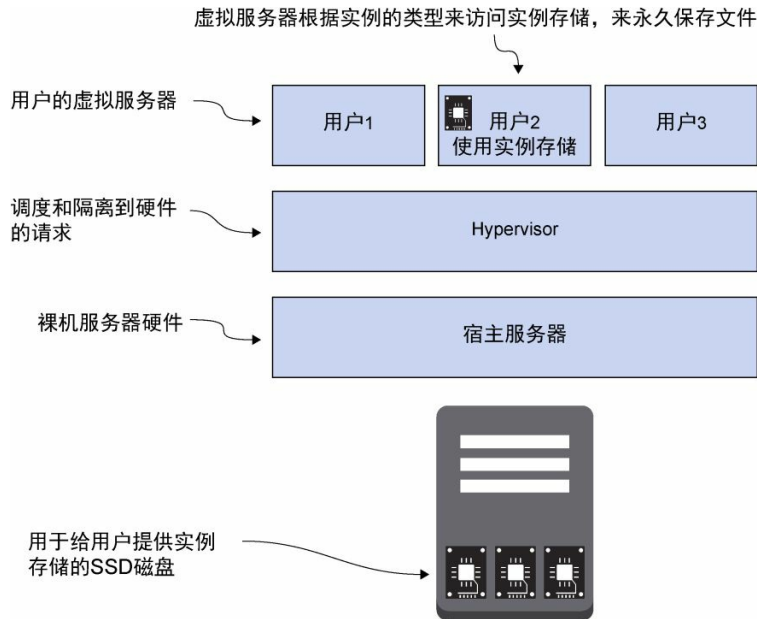


图8-2 实例存储是EC2实例的一部分

和通过网络挂载在EC2上的EBS卷不同，实例存储包含在虚拟服务器中，没有虚拟服务器就不再存在。

不要使用实例存储来存放不能丢失的数据，把它用来存放缓存、临时数据和一些在多个节点间复制数据的应用如某些数据库。如果用户想使用钟爱的NoSQL数据库，应用很有可能负责复制数据，这样用户可以安全地使用实例存储来获得最高的I/O性能。

### 警告

如果用户停止或者终结自己的EC2实例，实例存储上的数据会丢失。这意味着用户的数据会被删除并且无法恢复！

AWS提供SSD和物理磁盘的实例存储，容量从4 GB到48 TB不等。表8-3列出了所有提供实例存储的EC2实例类型。

表8-3 提供实例存储的实例类型

使用场景	实例类型	实例存储类型	实例存储容量（GB）
通用类型	m3.medium–m3.2xlarge	SSD	1 × 4~2 × 80
优化的计算	c3.large–c3.8xlar	SSD	2 × 16~2 × 320
优化的内存	r3.large–r3.8xlar	SSD	1 × 32~2 × 320
优化的存储	i2.xlarge–i2.8xlarge	SSD	1 × 800~8 × 800
优化的存储	d2.xlarge–d2.8xlarge	HDD	3 × 2 000~24 × 2 000

如果希望手动创建一个提供实例存储的EC2实例，按照3.1.1节中的步骤打开AWS控制台。运行启动EC2实例的向导。

**警告**

启动m3.medium的虚拟服务器将产生费用。如果想了解当前的每小时费用，可以访问AWS官方网站。

- 完成第1步到第3步：选择一个AMI，选择m3.medium实例类型，并配置实例详细信息。
- 在第4步，如图8-3所示那样配置实例存储。
  - 点击“添加新卷”按钮。
  - 选择“实例存储0”。
  - 设置设备名为“/dev/sdb”。
- 完成第5步到第7步：为实例打标签，配置安全组，检查并启动实例。

现在你的EC2实例可以使用实例存储了。

代码清单8-1展示了如何使用CloudFormation来使用实例存储。如果用户启动EBS为根卷的EC2实例（这是默认情况），用户必须定义BlockDeviceMappings来映射EBS卷和实例存储到特定的设备名。和创建EBS卷的模板不同，实例存储不是标准的独立资源；实例存储是EC2的一部分：根据实例类型的不同，可以选择零个、1个或者多个实例存储作映射。



图8-3 在启动EC2实例时添加实例存储

代码清单8-1 使用CloudFormation创建连接实例存储的EC2实例

```
"Server": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    "InstanceType": "m3.medium",      <--- 选择提供实例存储的实例类型
    [...]
    "BlockDeviceMappings": [{
      "DeviceName": "/dev/xvda",      <--- EBS 根卷（存放操作系统文件）
      "Ebs": {
        "VolumeSize": "8",
        "VolumeType": "gp2"
      }
    }, {
      "DeviceName": "/dev/xvdb",      <--- 实例存储会显示为/dev/xvdb 设备文件
      "VirtualName": "ephemeral0"    <--- 实例存储的虚拟名称为ephemeral0或者ep
hemeral1
    }
  ]
}
```

### 基于Windows操作系统的EC2实例

BlockDeviceMapping同样适用于Windows操作系统。设备名和分区字符（如C:/、D:/等）不同。DeviceName要想变成分区字符，首先要把卷挂载在EC2上。代码清单8-1中显示的实例存储必须被挂载为Z:/。继续阅读以了解Linux操作系统的步骤。

### 资源清理

完成本部分之后要删除手动启动的EC2实例，以清除用过的资源，否则将会为创建的资源付费。

## 8.2.1 使用实例存储

为了帮助读者使用实例存储，我们创建了CloudFormation模板，并存储在[https://s3.amazonaws.com/awsinaction/chapter8/instance\\_store.json](https://s3.amazonaws.com/awsinaction/chapter8/instance_store.json)。

### 警告

启动带有实例存储的m3.medium实例将产生费用。如果想了解当前的价格信息，可以访问AWS官方网站。

使用该模板创建一个堆栈，复制PublicName 的输出，并使用SSH登录到EC2实例。你可以使用fdisk 命令查看挂载的实例存储。通常，实例存储可以在/dev/xvdb到/dev/xvde设备文件中找到。

```
$ sudo fdisk -l
Disk /dev/xvda: 8589 MB [...]    <--根卷（存放操作系统文件）
Units = Sektoren of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: gpt

#      Start          End      Size Type              Name
  1      4096      16777182      8G  Linux filesystem Linux
128      2048          4095      1M  BIOS boot parti BIOS Boot Partition
```



```
Disk /dev/xvdb: 4289 MB [...]    <---实例存储
Units = Sektoren of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

要查看挂载的卷，运行下面的命令：

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/xvda1      7.8G  1.1G   6.6G  14% /          <---根卷（存放操作系统）
devtmpfs        1.9G   60K   1.9G   1% /dev
tmpfs           1.9G    0   1.9G   0% /dev/shm
/dev/xvdb       3.9G  1.1G   2.7G  28% /media/ephemeral0    <---实例存储自动挂载
```

实例存储将自动挂载到/media/ephemeral0目录下。如果EC2实例有多个实例存储，将分别挂载到ephemeral1、ephemeral2等目录。接着我们来进行一些性能测试。

## 8.2.2 性能测试

下面使用相同的性能测试来比较实例存储和EBS卷：

```
$ sudo dd if=/dev/zero of=/media/ephemeral0/tempfile bs=1M count=1024 \
conv=fdatasync,notrunc

1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 5.93311 s, 181 MB/s    <---EBS 的3 倍的读
性能
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3

$ sudo dd if=/media/ephemeral0/tempfile of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 3.76702 s, 285 MB/s    <---EBS 的4 倍的写
```

性能

根据实际负载的不同，性能可能有所差异。本示例中使用1MB大小的文件。如果服务的是一个互联网站点，很有可能你将处理大量的小文件。但是这个性能测试显示实例存储就像一个普通的磁盘，性能也和一个普通的磁盘相近。

#### 资源清理

在本节结束时别忘了删除堆栈来清除所有用过的资源，否则很可能会因为使用这些资源被收取费用。

### 8.2.3 备份数据

实例存储卷没有内建的方法来进行备份。利用在7.2节所学的知识，可以使用Cron和S3来定期备份数据：

```
$ aws s3 sync /path/to/data s3://$YourCompany-backup/serverdata
```

如果需要备份实例存储的数据，很可能更持久的块存储EBS卷会是更合适的选择。实例存储更适合对数据持久化要求不高的数据。

### 8.3 比较块存储解决方案

表8-4展示了S3、EBS和实例存储的区别。用户可以参考表决定哪种存储最适合自己的应用。基本原则是：如果应用支持S3，就使用S3；否则就选择EBS。

表8-4 S3和AWS块存储方案的比较

	S3	EBS	实例存储
常见的使用场景	集成到应用程序中以保存用户上传的内容	为需要块级别存储的数据库或者传统应用程序提供持久化	提供临时数据存储或者为内建复制技术来防止数据丢失的应用程序提供高性能存储
独立的资源	是	是	否
如何访问数据	HTTPS API	EC2实例/系统调用	EC2实例/系统调用
是否有文件系统	没有	有	有
防止数据丢失	很高	高	低
每GB容量成本	\$\$	\$\$\$	\$

运维 开销	无	低	中等
----------	---	---	----

下面来看一个真实世界里使用实例存储和EBS卷的例子。

## 8.4 使用实例存储和EBS卷提供共享文件系统

仅使用AWS提供的块级别存储解决方案无法解决下面的问题：如何同时在多个EC2实例之间共享块存储。用户可以使用网络文件系统（Network File System，NFS）协议来解决这个问题。

### Amazon Elastic File System已经发布\*\*

AWS提供了一款名为Amazon Elastic File System的服务。EFS是一个分布式的文件系统服务，基于网络文件系统第4版（Network File System Version，NFSv4）协议。读者可用选择它来解决在多台服务器之间共享块数据的需求。读者可以访问AWS的官方网站，了解EFS是否可以在你选择的区域里使用EFS。

图8-4展示了一台EC2实例如何工作为一台NFS服务器，并且通过NFS共享文件。其他EC2实例（NFS客户端）可以通过网络连接从NFS服务器挂载NFS共享。为了改善延时的性能，NFS服务器上使用了实例存储。但是你已经了解到实例存储无法提供很好的持久性，所以必须保护数据。NFS服务器上还挂载了一个EBS卷，数据将定期同步到EBS卷上。最坏的情况是丢失自上一次同步操作后改变的数据。在某些情况下（例如，在Web服务器之间共享PHP文件），这些数据丢失是可以接受的，因为数据可以被重新加载。

### NFS服务器是单点故障

设置NFS服务器可能不适合业务关键性生产环境。NFS服务器是一个单点故障：如果EC2实例故障，其他NFS客户端将无法访问共享文件。请慎重选择使用共享的文件系统。在多数情况下，如果应用数据变化量不大，S3会是一个很好的选择。如果真的需要一个共享的文件系统，考虑使用Amazon EFS或者设置GlusterFS。

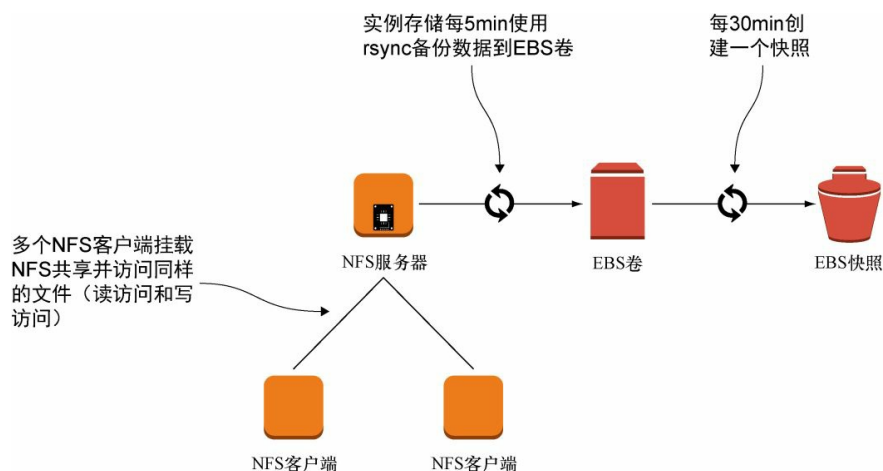


图8-4 NFS可以用来在EC2实例之间共享块级别存储

我们创建一个CloudFormation模板和Bash脚本来把这个系统拓扑付诸实现。需要按顺序完成下面的步骤：

- （1）添加安全组以保证NFS的安全。
- （2）添加NFS服务器的EC2实例和EBS卷。
- （3）创建安装和配置脚本到NFS服务器。
- （4）添加NFS客户端的EC2实例。

下面开始具体的操作。

## 8.4.1 NFS的安全组

如何控制应用程序之间的网络访问？在设计安全组的时候必须回答这个问题。为了简化问题（将省略堡垒主机的创建过程），所有EC2实例将允许来自互联网（0.0.0.0/0）的SSH访问。NFS服务器必须能够通过NFS所需要的端口访问到（TCP和UDP协议：111，2049），但是只有客户端才应该有对这些端口访问，如代码清单8-2所示。

代码清单8-2 NFS服务配置安全组

```

"SecurityGroupClient": {      <--- 安全组关联到NFS 客户端。这个安全组不包含任何规
  
```

则：仅用来标记来自客户端的流量

```
"Type": "AWS::EC2::SecurityGroup",
"Properties": {
  "GroupDescription": "My client security group",
  "VpcId": {"Ref": "VPC"}
}
},

"SecurityGroupServer": {      <---关联到NFS 服务器的安全组
  "Type": "AWS::EC2::SecurityGroup",
  "Properties": {
    "GroupDescription": "My server security group",
    "VpcId": {"Ref": "VPC"},
    "SecurityGroupIngress": [{
      "SourceSecurityGroupId": {"Ref": "SecurityGroupClient"},      <--- 允许
来自NFS 客户端（选择客户端安全组作为允许来源）的所有入站访问（TCP 协议）
      "IpProtocol": "tcp",
      "FromPort": 111,
      "ToPort": 111
    }, {
      "SourceSecurityGroupId": {"Ref": "SecurityGroupClient"},
      "IpProtocol": "udp",      <--- 允许访问端口111（UDP 协议）
      "FromPort": 111,
      "ToPort": 111
    }, {
      "SourceSecurityGroupId": {"Ref": "SecurityGroupClient"},
      "IpProtocol": "tcp",      <--- 允许来自NFS 客户端的入站流量访问nfsd 服务的
端口2049
      "FromPort": 2049,
      "ToPort": 2049
    }, {
      "SourceSecurityGroupId": {"Ref": "SecurityGroupClient"},
      "IpProtocol": "udp",
      "FromPort": 2049,
      "ToPort": 2049
    }
  ]
}
},

"SecurityGroupCommon": {      <---关联到NFS 服务器和客户端的通用安全组
  "Type": "AWS::EC2::SecurityGroup",
  "Properties": {
    "GroupDescription": "My security group",
    "VpcId": {"Ref": "VPC"},
    "SecurityGroupIngress": [{      <--- 允许来自互联网的SSH 入站流量
      "CidrIp": "0.0.0.0/0",
      "FromPort": 22,
      "IpProtocol": "tcp",
```

```

        "ToPort": 22
      }]
    }
  }
}

```

有趣的是**SecurityGroupClient** 没有定义任何规则。它只用来标记来自NFS客户端的访问。**SecurityGroupServer** 使用**SecurityGroupClient** 识别被允许NFS客户端访问的源地址。

## 8.4.2 NFS服务器和卷

NFS服务器的实例类型必须提供一个实例存储。本例将用到**m3.medium**实例，因为它是自带实例存储的最便宜的实例类型，虽然它只提供4 GB容量。如果需要更大的容量，你必须选择其他的实例类型。这台服务器关联了两个安全组：**SecurityGroupCommon** 允许SSH访问，**SecurityGroupServer** 允许NFS相关的端口访问。这台服务器必须在启动的时候安装和配置NFS服务，所有你将用到一个bash脚本；你将在下面的步骤中创建该脚本。使用bash脚本提供更好的可读性——因为有时**UserData** 格式很烦琐。为了防止数据丢失，将创建一个EBS卷来作为实例存储的备份，如代码清单8-3所示。

代码清单8-3 NFS服务器和卷

```

"Server": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    "IamInstanceProfile": {"Ref": "InstanceProfile"},
    "ImageId": "ami-1ecae776",
    "InstanceType": "m3.medium",      <--m3.medium 提供4GB容量的SSD 实例存储
    "KeyName": {"Ref": "KeyName"},
    "SecurityGroupIds": [{"Ref": "SecurityGroupCommon"},
      {"Ref": "SecurityGroupServer"}],  <--使用服务器的安全组来过滤网络访问
    "SubnetId": {"Ref": "Subnet"},
    "BlockDeviceMappings": [{
      "Ebs": {      <--映射根EBS 卷到/dev/xvda
        "VolumeSize": "8",
        "VolumeType": "gp2"
      }
    }]
  }
}

```



```

    },
    "DeviceName": "/dev/xvda"
  }, {
    "VirtualName": "ephemeral0",      <--映射实例存储到/dev/xvdb
    "DeviceName": "/dev/xvdb"
  }],
  "UserData": { "Fn::Base64": { "Fn::Join": [ "", [
    "#!/bin/bash -ex\n",
    "curl -s https://[...]/nfs-server-install.sh | bash -ex\n"      <--下
    载并执行安装脚本（仅从可信任来源下载！）
  ] ] } }
}
},
"Volume": {
  "Type": "AWS::EC2::Volume",      <--创建5GB 的备份存储（容量足够备份4GB 的实
  例存储）
  "Properties": {
    "AvailabilityZone": { "Fn::GetAtt": [ "Server", "AvailabilityZone" ] },
    "Size": "5",
    "VolumeType": "gp2"
  }
},
"VolumeAttachment": {
  "Type": "AWS::EC2::VolumeAttachment",      <--将卷附加到服务器（至/dev/xvdf
  ）
  "Properties": {
    "Device": "/dev/xvdf",
    "InstanceId": { "Ref": "Server" },
    "VolumeId": { "Ref": "Volume" }
  }
}
}

```

现在你可以在启动的时候安装和配置NFS服务器了。

### 8.4.3 NFS服务器安装和配置脚本

为了运行NFS，用户需要使用yum 安装相关的软件并且配置和启动服务。为了定期备份实例存储的数据，用户还需要挂载EBS卷并且定时运行cron作业来复制数据到EBS卷。最后，将从EBS卷创建一个EBS快照。安装和配置脚本如代码清单8-4所示。

```
#!/bin/bash -ex

yum -y install nfs-utils nfs-utils-lib      <--安装NFS 软件包
service rpcbind start      <--启动rpcbind 进程（NFS 的依赖）
service nfs start      <--启动NFS 进程
chmod 777 /media/ephemeral0      <--允许用户读写实例存储卷
echo "/media/ephemeral0 *(rw,async)" >> /etc/exports      <--使用NFS 导出实例存储卷给其他的NFS 客户端
exportfs -a      <--重新加载以应用修改后的配置

while ! [ "$(fdisk -l | grep '/dev/xvdf' | wc -l)" -ge "1" ]; \      <--挂载EBS 卷
do sleep 10; done

if [[ "$(file -s /dev/xvdf)" != *"ext4"* ]]      <--如果还不是ext4 文件格式，
则格式化EBS 卷（第一次启动服务器时进行该操作）
then
    mkfs -t ext4 /dev/xvdf
fi

mkdir /mnt/backup
echo "/dev/xvdf /mnt/backup ext4 defaults,nofail 0 2" >> /etc/fstab
mount -a      <--等待EBS 卷创建完成

INSTANCEID=$(curl -s http://169.254.169.254/latest/meta-data/instance-id)
VOLUMEID=$(aws --region us-east-1 ec2 describe-volumes \      <--获得EBS卷的ID
--filters "Name=attachment.instance-id,Values=$INSTANCEID" \
--query "Volumes[0].VolumeId" --output text)

cat > /etc/cron.d/backup << EOF      <--在cron 作业的定义中复制到EOF 的所有文本。在/etc/cron.d/目录中保存cron 作业的定义

SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/opt/aws/bin      <--确保/opt/aws/bin 在执行路径中，以方便运行AWS 命令
MAILTO=root
HOME=/
*/15 * * * * root rsync -av --delete /media/ephemeral0/ /mnt/backup/ ; \
    <--每15 min从实例存储卷同步数据到EBS 卷
fsfreeze -f /mnt/backup/ ; \      <--冻结EBS 卷以创建一致的快照
aws --region us-east-1 ec2 create-snapshot --volume-id $VOLUMEID ; \      <--解冻EBS 卷
fsfreeze -u /mnt/backup/      <--创建EBS 快照
EOF
```

因为脚本会通过命令行工具调用AWS API，EC2实例需要权限来调用这些API。我们将通过使用IAM角色的方式来给EC2进行授权，如代码清单8-5所示。

代码清单8-5 IAM角色

```
"InstanceProfile": {
  "Type": "AWS::IAM::InstanceProfile",      <--在NFS 服务器上附加IAM 配置文件
  "Properties": {
    "Path": "/",
    "Roles": [{"Ref": "Role"}]
  }
},
"Role": {
  "Type": "AWS::IAM::Role",      <--定义IAM 角色
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [{
        "Effect": "Allow",
        "Principal": {
          "Service": ["ec2.amazonaws.com"]
        },
        "Action": ["sts:AssumeRole"]
      }]
    },
    "Path": "/",
    "Policies": [{
      "PolicyName": "ec2",
      "PolicyDocument": {
        "Version": "2012-10-17",
        "Statement": [{
          "Sid": "Stmt1425388787000",
          "Effect": "Allow",
          "Action": ["ec2:DescribeVolumes", "ec2:CreateSnapshot"],      <--
在大量小文件时使用rsync 工具
          "Resource": ["*"]
        }]
      }
    }]
  }
}
```

授权允许描述卷和创建快照的操作

如果用户的场景中需要访问海量的小文件（超过100万个小文件），rsync 将花费大量的时间，并且消耗CPU周期。可以考虑使用DRBD来异步地从实例存储同步数据到EBS卷。设置过程会稍微复杂些（如果使用的是Amazon Linux），但是可以获得更好的性能。

还有一个操作没有完成：配置客户端。我们稍后将完成。

## 8.4.4 NFS客户端

NFS共享可以被多个客户端挂载。为了演示，使用两个客户端Client1 和Client2 就足够了。Client2 是Client1 的副本。NFS客户端代码如代码清单8-6所示。

代码清单8-6 NFS客户端

```
"Client1": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    "ImageId": "ami-1ecae776",
    "InstanceType": "t2.micro",
    "KeyName": {"Ref": "KeyName"},
    "SecurityGroupIds": [{"Ref": "SecurityGroupCommon"},      <-- 将通用安全
组与客户端安全组联系起来
    {"Ref": "SecurityGroupClient"}],
    "SubnetId": {"Ref": "Subnet"},
    "UserData": {"Fn::Base64": {"Fn::Join": ["", [
      "#!/bin/bash -ex\n",
      "yum -y install nfs-utils nfs-utils-lib\n",
      "mkdir /mnt/nfs\n",
      "echo \"\", {\"Fn::GetAtt\": [\"Server\", \"PublicDnsName\"]}],      <-- 将NFS
共享条目写入fstab
      \":/media/ephemeral0 /mnt/nfs nfs rw 0 0\" >> /etc/fstab\n",
      "mount -a\n"      <-- 挂载NFS 共享
    ]}}}
  }
}
```

现在可以体验一下如何通过NFS来共享文件。

## 8.4.5 通过NFS共享文件

为了帮助读者研究NFS，本书提供了一个CloudFormation模板，位于<https://s3.amazonaws.com/awsinaction/chapter8/nfs.json>。

### 警告

启动实例类型为m3.medium的虚拟服务器将产生费用。如果想了解当前每小时的价格信息，可以访问AWS官方网站。

使用该模板创建一个堆栈，在输出中复制**Client1PublicName** 字段，通过SSH登录到主机。

在/mnt/nfs/目录下创建文件：

```
$ touch /mnt/nfs/test1
```

现在，在堆栈输出中复制**Client2PublicName** 并SSH登录到第二台客户端。在/mnt/nfs/下列出所有文件：

```
$ ls /mnt/nfs/  
test1
```

太棒了。我们现在可以在多台EC2实例之间共享文件了。

### 资源清理

在本节结束时别忘了删除堆栈来清除所有用过的资源，否则很可能会因为使用这些资源被收取费用。

## 8.5 小结

- 数据块存储可用配合EC2实例来使用，因为操作系统需要块级别的存储（包括分区、文件系统和读/写系统调用）。
- EBS卷通过网络连接到EC2实例。根据实例类型的不同，网络连接的带宽也不同。
- EBS快照功能提供了强大的工具来把EBS卷的数据备份到S3，因为它们使用的是数据块级别的增量的复制方式。
- 实例存储是EC2实例的一部分，快速且廉价。但是在EC2实例停止或者终结的时候，实例存储上的数据会丢失。
- 可以使用NFS在EC2实例之间共享文件。

## 第9章 使用关系数据库服务：RDS

### 本章主要内容

- 使用RDS来启动和初始化关系数据库
- 创建和使用快照来恢复数据库
- 设置高可用的数据库
- 调整数据库的性能
- 监控数据库

关系数据库是业界存储和查询结构化数据的事实上的标准，许多应用程序都搭建在MySQL、Oracle数据库、微软SQL Server或者PostgreSQL这样的关系数据库上。典型的关系数据库专注提供数据一致性和保证ACID的数据库事务（原子性、一致性、隔离性和持久性）的功能。关系数据库的典型任务是在财务应用中存储和查询像账户、交易这样的结构化的数据。

如果想在AWS上使用一个关系数据库，有以下两个选择。

- 使用托管的数据库服务，如Amazon RDS，由AWS提供。
- 在虚拟服务器上自己搭建关系数据库。

亚马逊关系数据库服务（Amazon RDS）提供了方便、可用的关系数据库。在底层，Amazon RDS运行一个常见的关系数据库。在本书编写的时候，它支持MySQL、Oracle Database、微软SQL Server和PostgreSQL<sup>[1]</sup>。如果应用程序使用上述的几种关系数据库的引擎，迁移到Amazon RDS将非常容易。

<sup>[1]</sup> 现在额外添加了对Amazon Aurora 和MariaDB 的支持。——译者注

#### Amazon Aurora已经发布

AWS发布了一款新的数据库引擎叫作Amazon Aurora。Aurora兼容MySQL，但是以更低的成本提供更好的可用性和性能。你可以用Aurora来替代MySQL。访问AWS官方网站，可以了解更多信息。

RDS是一个托管的服务。托管服务由服务提供商进行运维管理——在Amazon RDS中由AWS管理。托管服务提供商负责提供一系列的服务——Amazon RDS服务负责关系数据库的运维管理。表9-1比较了使用RDS数据库和在虚拟服务器上自行搭建数据库的区别。

表9-1 托管服务RDS和虚拟服务器上自建数据库的比较

	Amazon RDS	在虚拟服务器上自己搭建
AWS服务的成本	更高，因为RDS的成本高于EC2虚拟服务器	更低，因为EC2虚拟服务器比RDS便宜
总体拥有成本	更低，因为更多客户分摊了运维成本	高很多，因为需要人力来管理数据库
质量	AWS专业人员负责托管服务	你需要搭建专业团队和进行质量管理
灵活性	高，因为你可以选择数据库引擎和修改配置参数	更高，因为你可以控制安装在虚拟服务器上的数据库的每个部分

在虚拟服务器上搭建关系数据库需要大量的时间和技能，所以推荐在尽可能的情况下，使用Amazon RDS提供需要的关系数据库，以降低成本和改善质量。这是为什么在本书中不会介绍如何在EC2上自己搭建关系数据库。相反的，我们将介绍Amazon RDS的细节。

在本章中，我们将使用Amazon RDS启动一个MySQL数据库。第2章介绍的WordPress搭建使用图9-1所示的架构，在本章中我们将再次使用这个示例，但是这次专注于数据库的部分。在基于Amazon RDS的MySQL数据库运行起来以后，你将了解如何导入数据，备份和恢复数据。更高级的话题，如搭建高可用的数据库和改善性能的内容将随后介绍。



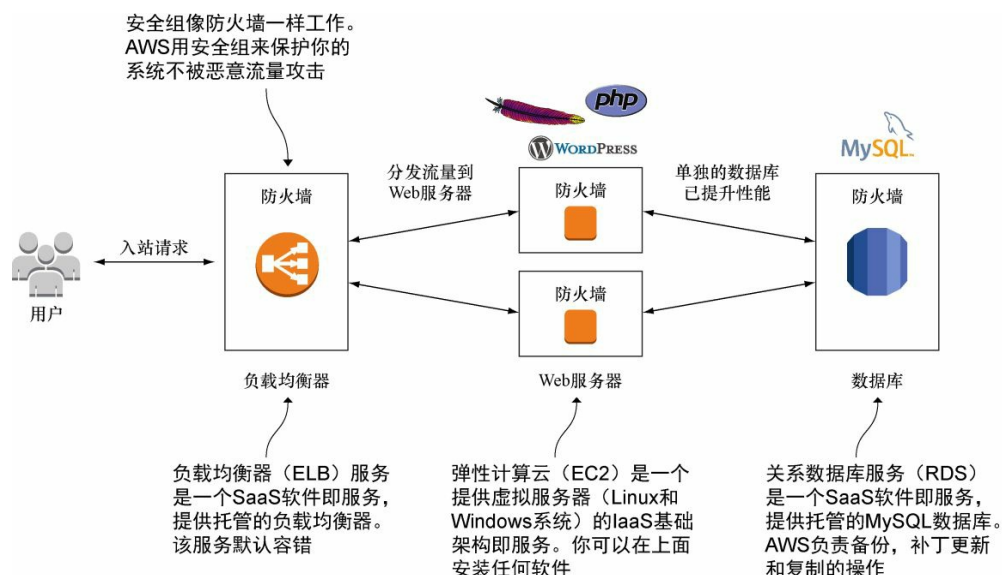


图9-1 公司的博客系统包含两个负载均衡的运行WordPress的Web服务器和一台运行MySQL数据库的服务器

#### 示例都包含在免费套餐中

本章中的所有示例都包含在免费套餐中。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

本章中的示例使用一个用于WordPress应用的MySQL数据库。用户可以轻松地把学到的知识应用在其他如Oracle数据库、微软SQL Server和PostgreSQL，以及WordPress以外的其他应用程序。

## 9.1 启动一个MySQL数据库

流行的博客平台WordPress搭建在MySQL关系数据库上。如果你想在自己的服务器上搭建博客，则需要运行PHP应用程序（例如，安装Apache Web服务器），并且需要操作一个MySQL数据库，以存放WordPress的文章、评论和作者信息。Amazon RDS以服务的方式提供MySQL数据库。你不再需要自己安装、配置和操作MySQL数据库。

### 9.1.1 用Amazon RDS数据库启动WordPress平台

启动一个数据库包含两个步骤。

- （1）启动一个数据库实例。
- （2）连接应用程序到数据库的端点。

我们需要使用第2章中的CloudFormation模板来创建WordPress博客平台和MySQL数据库。你还将在模板里使用Amazon RDS服务。你可以在下载的源代码中找到该模板，我们要用的模板位于chapter9/template.json。同样的文件也保存在S3上下面的位置[https://s3.amazonaws.com/ awsinaction/chapter9/template.json](https://s3.amazonaws.com/awsinaction/chapter9/template.json)。

执行下面的命令来创建包含一个MySQL引擎的RDS实例和服务WordPress应用的Web服务器：

```
$ aws cloudformation create-stack --stack-name wordpress --template-url \
https://s3.amazonaws.com/awsinaction/chapter9/template.json \
--parameters ParameterKey=KeyName,ParameterValue=mykey \
ParameterKey=AdminPassword,ParameterValue=test1234 \
ParameterKey=AdminEmail,ParameterValue=your@mail.com
```

CloudFormation堆栈的创建大概需要几分钟的时间，所以你有充分

的时间来了解RDS数据库实例的细节。代码清单9-1给出了用于创建wordpress 堆栈的CloudFormation模板的一些组件。

陷阱：媒体上传和插件

WordPress使用MySQL数据库来保存文章和用户信息。但是默认情况下，WordPress在本地文件系统的wp-content目录存储用户上传的媒体文件和插件。这样的架构不是无状态的设计。这种架构下你无法使用多台服务器提供服务，因为只有一台服务器上保存有用户上传的媒体文件和插件。

本章中的示例并不完整，因为它没有解决上面提到的问题。如果对如何解决这个问题感兴趣，可以查看第14章的内容。第14章将介绍通过自动化配置虚拟服务器的方式来自动安装插件，并且将把媒体文件上传到对象存储进行保存。

表9-2展示了在管理控制台上使用CloudFormation创建RDS数据库所需要的一些属性。

表9-2 创建RDS数据库需要的属性

属 性	描 述
AllocatedStorage	数据库的容量（以GB为单位）
DBInstanceClass	底层虚拟服务器的实例类型
Engine	数据库引擎（MySQL、Oracle数据库、Microsoft SQL服务器或者PostgreSQL）
DBInstanceIdentifier	数据库实例的标识符
DBName	数据库的名字
MasterUsername	主用户的用户名
MasterUserPassword	主用户的密码

RDS数据库可以部署在一个虚拟的私有网络中（VPC）。推荐用户这么做以保护数据，而且不要部署一个公网IP地址给数据库。在VPC中部署RDS数据库的情况下，可以使用私有IP地址来和RDS实例通信。这样数据库不会直接在互联网上被访问到。如果想在VPC里部署RDS实例，需要指定数据库所在的子网，如代码清单9-1所示。

代码清单9-1 创建RDS数据库的CloudFormation模板代码片段

```
{
  [...]
  "Resources": {
    [...]
    "DatabaseSecurityGroup": {      <---数据库实例的安全组，允许来自Web服务器流量访问MySQL 默认端口
      "Type": "AWS::EC2::SecurityGroup",
      "Properties": {
        "GroupDescription": "awsinaction-db-sg",
        "VpcId": {"Ref": "VPC"},
        "SecurityGroupIngress": [{
          "IpProtocol": "tcp",      <---MySQL的默认端口是3306
          "FromPort": "3306",
          "ToPort": "3306",
          "SourceSecurityGroupId": {
            "Ref": "WebServerSecurityGroup"      <---引用Web 服务器所在的安全组
          }
        }]
      }
    },
    "Database": {
      "Type": "AWS::RDS::DBInstance",      <---创建亚马逊RDS数据库实例
      "Properties": {
        "AllocatedStorage": "5",      <---数据库提供5 GB 容量
        "DBInstanceClass": "db.t2.micro",      <---数据库服务器的实例类型是t2.micro，可选的最小的实例类型
        "DBInstanceIdentifier": "awsinaction-db",      <---RDS 数据库的标识符
        "DBName": "wordpress",      <---创建名为wordpress 的默认的数据库
        "Engine": "MySQL",      <---使用MySQL 作为数据库的引擎
        "MasterUsername": "wordpress",      <---MySQL 数据库的主用户的用户名
        "MasterUserPassword": "wordpress",      <---MySQL 数据库主用户的密码
        "VPCSecurityGroups": [
          {"Fn::GetAtt": ["DatabaseSecurityGroup", "GroupId"]}
        ],      <---引用数据库的安全组名称
        "DBSubnetGroupName": {
          {"Ref": "DBSubnetGroup"}      <---定义RDS 数据库实例将启动在哪个子网
        }
      }
    },
  },
}
```

```

    "DBSubnetGroup" : {
      "Type" : "AWS::RDS::DBSubnetGroup",      <-- 创建子网组来定义数据库实例所
    在的子网
      "Properties" : {
        "DBSubnetGroupDescription" : "DB subnet group",
        "SubnetIds": [
          {"Ref": "SubnetA"},
          {"Ref": "SubnetB"}
        ]      <-- 在子网A或者子网B 中启动RDS 数据库实例
      }
    },
    [...]
  },
  [...]
}

```

使用下面的命令检查名为wordpress 的CloudFormation堆栈是否进入CREATE\_COMPLETE 状态:

```
$ aws cloudformation describe-stacks --stack-name wordpress
```

在输出栏中查找StackStatus堆栈状态，并且查看是否状态已经为CREATE\_COMPLETE 创建完成。如果不是，再等待几分钟（创建堆栈可能需要多达15 min），然后再运行该命令。当状态已经为CREATE\_COMPLETE 时，你将在输出部分看到Outputkey 属性。对应的OutputValue 包含了WordPress博客平台的URL连接。代码清单9-2给出了详细的输出。在浏览器中打开该URL连接，你将看到一个正在运行的WordPress服务。

代码清单9-2 检查CloudFormation堆栈的状态

```

$ aws cloudformation describe-stacks --stack-name wordpress
{
  "Stacks": [{
    "StackId": "...",
    "Description": "AWS in Action: chapter 9",
    "Parameters": [{
      "ParameterValue": "mykey",
      "ParameterKey": "KeyName"
    }
  ]
}

```

```

    }],
    "Tags": [],
    "Outputs": [{
      "Description": "Wordpress URL",
      "OutputKey": "URL",
      "OutputValue": "http://[...].com/wordpress"    <--在浏览器中打开此URL
    }],
    "StackStatusReason": "",
    "CreationTime": "2015-05-16T06:30:40.515Z",
    "StackName": "wordpress",
    "NotificationARNs": [],
    "StackStatus": "CREATE_COMPLETE",    <--等待CloudFormation堆栈完成创建
    "DisableRollback": false
  }
}

```

超链接访问WordPress博客程序

启动和操作一个MySQL这样的数据库就是这么简单。当然，除了使用CloudFormation的模板外，你也可以使用管理控制台来启动一个RDS数据库实例模板。RDS是一个托管的服务，AWS负责大部分的操作任务来保证数据库是安全和可靠的。你只需要专注在下面的任务。

- 监控数据库的可用存储空间，确保在需要的时候增加存储空间。
- 监控数据库的性能，确保在需要的时候增加I/O性能和计算性能。

这两样工作都可以使用CloudWatch监控来帮助完成，稍后将了解这部分内容。

## 9.1.2 探索使用MySQL引擎的RDS数据库实例

CloudFormation堆栈创建了一个带MySQL引擎的RDS数据库。每个RDS数据库都提供了一个端点来接受SQL请求。应用程序可以发送请求到这个端点来查询和存储数据。使用**describe** 命令可以获得端点和其他的详细信息：

```
$ aws rds describe-db-instances
```

这个请求的输出包含了代码清单9-2所示的RDS数据库实例的详细信息。连接到RDS数据库所需的最重要的几个属性如表9-3所示。

表9-3 连接到RDS数据库所需要的属性

属 性	描 述
Endpoint	数据库端点的主机名和端口，以便应用程序连接到数据库。这个端点接受SQL命令
DBName	启动时自动创建的默认数据库的名字
MasterUsername	数据库主用户的用户名。这里没有显示密码；你必须记住密码或者在CoudFormation模板中查找。主用户可以创建额外的数据库用户。具体的步骤取决于具体的数据库引擎
Engine	描述该数据库实例使用的数据库类型。本例中是MySQL

这里有很多其他属性。你将在本章稍后了解到关于它们的更多的信息。代码清单9-3描述了MySQL关系数据库的实例。

代码清单9-3 描述MySQL RDS数据库实例

```
{
  "DBInstances": [{
    "PubliclyAccessible": false,      <---该数据库无法从互联网访问到-只可以从私有网络（VPC）访问
    "MasterUsername": "wordpress",    <---MySQL 数据库的主用户的用户名
    "LicenseModel": "general-public-license",
    "VpcSecurityGroups": [{
      "Status": "active",
      "VpcSecurityGroupId": "sg-7a84aa1e"    <---数据库的安全组，只允许Web 服务器访问3306 端口
    }],
    "InstanceCreateTime": "2015-05-16T06:40:33.107Z",
    "OptionGroupMemberships": [{
      "Status": "in-sync",
      "OptionGroupName": "default:mysql-5-6"    <---选项组用于额外的数据库相关的配置
    }]
  }]
}
```

```

    }],
    "PendingModifiedValues": {},
    "Engine": "mysql",      <--数据库实例运行的是MySQL 引擎
    "MultiAZ": false,      <--没有启用高可用的设置。在9.5 节你将了解如何设置
    "LatestRestorableTime": "2015-05-16T08:00:00Z",
    "DBSecurityGroups": [],
    "DBParameterGroups": [{
        "DBParameterGroupName": "default.mysql5.6",
        "ParameterApplyStatus": "in-sync"      <--参数组用于配置数据库引擎的参数
    }],
    "AutoMinorVersionUpgrade": true,      <--RDS 将自动进行数据库的小版本补丁
升级
    "PreferredBackupWindow": "06:01-06:31",      <--每天创建数据库快照的时间窗口（UTC 时间）
    "DBSubnetGroup": {
        "Subnets": [{      <--用于启动数据库实例的子网
            "SubnetStatus": "Active",
            "SubnetIdentifier": "subnet-f045c9db",
            "SubnetAvailabilityZone": {
                "Name": "us-east-1a"
            }
        }], {
            "SubnetStatus": "Active",
            "SubnetIdentifier": "subnet-42e4a235",
            "SubnetAvailabilityZone": {
                "Name": "us-east-1b"
            }
        }],
    "DBSubnetGroupName": "wordpress-dbsubnetgroup-1lbc2t9palsej",
    "VpcId": "vpc-941e29f1",      <--启动数据库实例的私有网络（VPC）
    "DBSubnetGroupDescription": "DB subnet group",
    "SubnetGroupStatus": "Complete"
},
    "ReadReplicaDBInstanceIdentifiers": [],      <--RDS 允许为某些数据库实例
创建读副本。这将在9.6 节介绍
    "AllocatedStorage": 5,      <--数据库分配了5 GB 的存储容量。用户可以需要的时候增加容量
    "BackupRetentionPeriod": 1,      <--数据库快照备份将保留1 天
    "DBName": "wordpress",      <--默认数据库的名称
    "PreferredMaintenanceWindow": "mon:06:49-mon:07:19",      <--RDS 执行数据库引擎的小版本补丁升级的时间窗口（每周一6:49 到07:19, UTC 时间）
    "Endpoint": {      <--数据库实例的端点，应用程序用来发送SQL 请求。本例中是一个私有IP 地址，因为该数据库无法从互联网访问
        "Port": 3306,

```



```
    "Address": "awsinaction-db.czwgnecjynmj.us-east-1.rds.amazonaws.com"
  },
  "DBInstanceStatus": "available",      <-- 数据库的状态
  "EngineVersion": "5.6.22",           <-- 数据库引擎的版本为MySQL 5.6.22
  "AvailabilityZone": "us-east-1b",     <-- 数据库实例运行的数据中心
  "StorageType": "standard",            <-- 存储类型为标准类型，即物理磁盘。你将在后
续章节了解SSD 固态硬盘和预配置IOPS 性能的存储选项。
  "DbiResourceId": "db-SVHSQQOW4CPNR57LYLFXVHYOVU",
  "CACertificateIdentifier": "rds-ca-2015",
  "StorageEncrypted": false,            <-- 表示数据写入磁盘前是否进行加密
  "DBInstanceClass": "db.t2.micro",     <-- 数据库运行的虚拟服务器的实例类型
。db.t2.micro 是可选的最小的类型
  "DBInstanceIdentifier": "awsinaction-db" <-- 数据库实例的标识符
}]
}
```

RDS数据库在运行，但是它将产生多少成本？

### 9.1.3 Amazon RDS的定价

Amazon RDS数据库的定价取决于底层虚拟服务器的类型和分配的存储容量。和一个虚拟服务器（EC2）上运行的数据库相比，大概要额外付出30%的成本。在我们看来，Amazon RDS服务值得额外的成本，因为你不再需要担心典型的DBA任务，如安装、打补丁、升级、迁移、备份和恢复。Forester分析结果显示1个数据库管理员大概需要花费一半的时间来完成这些任务。

表9-4展示了在美国弗吉尼亚北区域的中等规模的RDS数据库实例的价格，该价格不包括高可用的故障切换功能。

表9-4 中等规模的RDS实例的月度成本

描 述	月度价格（美元）
数据库实例类型db.m3.medium	65.88

50 GB的通用类型SSD	5.75
额外的数据库快照容量（300 GB）	28.50
总计	100.13

我们已经为WordPress互联网应用启动了一个RDS数据库实例，下面来了解一下如何将数据导入RDS数据库。

## 9.2 将数据导入数据库

没有数据的数据库毫无用处。通常你需要给新的数据库导入数据。在从自有机房迁移到AWS的时候，还需要迁移数据库的数据。本部分将指导如何从MySQL数据库dump数据到使用MySQL引擎的RDS数据库。这一流程和其他所有数据库引擎（Oracle数据库、微软SQL Server、PostgreSQL）的迁移流程也相似。

要从数据库中将数据导入Amazon RDS数据库要按照下面的步骤操作。

- （1）导出自有数据中心里的数据库。
- （2）在RDS数据库所在的区域的同一个VPC中启动一台虚拟服务器。
- （3）把数据库导出的dump文件上传到该虚拟服务器。
- （4）从虚拟服务器中导入数据到RDS数据库。

我们将略过导出MySQL数据库数据的具体步骤。下面的内容介绍了导出现有的MySQL数据库的一些方法。

### 导出一个MySQL数据库

MySQL和其他所有的数据库系统都提供导出和导入数据库的方法。我们推荐使用MySQL提供的命令行工具来导出和导入数据库。你可能需要安装MySQL客户端工具。

下面的命令从本机导出所有数据库，并且把它们转储到名为dump.sql的文件。需要将\$UserName 替换为MySQL的admin或者master用户。

```
$ mysqldump -u $UserName -p --all-databases > dump.sql
```

还可以仅导出特定的数据库。如果有这样的需求，替换\$DatabaseName 为你想要导出的数据库的名称：

```
$ mysqldump -u $UserName -p $DatabaseName > dump.sql
```

当然也可以通过网络连接来导出数据库。要连接一个数据库来导出数据，替换\$Host 为主机名或者数据库的IP地址。

```
$ mysqldump -u $UserName -p $DatabaseName --host $Host > dump.sql
```

如果需要了解mysqldump 的更多信息，可查看MySQL文档。

理论上讲，你可以从任何自有数据中心的服务器或者本地网络导入数据库到RDS。但是通过互联网或者VPN连接的高延时将显著拖慢导入的过程。因此推荐进行额外的步骤：把数据库的转储文件上传到和RDS数据库位于相同区域和VPC里的虚拟服务器上，然后从那里导入数据库到RDS。

要完成这些操作，需要按照下面的步骤操作。

- （1）获得能够访问RDS数据库的虚拟服务器的公网IP（运行WordPress应用的虚拟服务器）。
- （2）通过SSH连接到该虚拟服务器。
- （3）从S3下载数据库的dump文件到虚拟服务器。
- （4）从RDS数据库运行import导入命令从虚拟服务器导入数据库到RDS。

幸运的是，你已经启动了两台可以连接到RDS上的MySQL数据库的虚拟服务器。在本机上运行下面的命令获取这两台虚拟服务器的公网IP地址：

```
$ aws ec2 describe-instances --filters Name=tag-key,\
Values=aws:cloudformation:stack-name Name=tag-value,\
Values=wordpress --output text \
--query Reservations[0].Instances[0].PublicIpAddress
```

建立SSH连接到该虚拟服务器。使用SSH的密钥mykey 来认证身

份，并且替换**\$PublicIpAddress** 为运行WordPress应用程序的虚拟服务器的公网IP：

```
$ ssh -i $PathToKey/mykey.pem ec2-user@$PublicIpAddress
```

作为示例，我们准备了一个用于WordPress博客的MySQL数据库的转储文件。使用下面的命令从S3上下载该转储文件。

```
$ wget https://s3.amazonaws.com/awsinaction/chapter9/wordpress-import.sql
```

现在已经就绪，可以开始把包含WordPress博客数据的转储文件导入到RDS数据库实例。你将需要RDS数据库实例上的MySQL数据库的端口，主机名（也叫作端点）。无法找到端点的信息？下面的命令可以帮助列出RDS数据库的端点。在本机上运行该命令：

```
$ aws rds describe-db-instances --query DBInstances[0].Endpoint
```

在虚拟服务器上运行下面的命令把wordpress-import.sql文件导入到RDS数据库实例；替换**\$DBHostName** 为之前的命令中列出的RDS的端点。当被提示输入密码的时候输入w ordpress：

```
$ mysql --host $DBHostName --user wordpress -p < wordpress-import.sql
```

在浏览器中在此访问WordPress博客，你将看到很多新的发帖和评论。如果无法找到博客的URL的话，在本机输入下面的命令来重新获取地址：

```
$ aws cloudformation describe-stacks --stack-name wordpress \  
--query Stacks[0].Outputs[0].OutputValue --output text
```

## 9.3 备份和恢复数据库

Amazon RDS是一个托管的服务，但是你仍然需要备份数据，这样在某些情况下或者被某些人损坏了数据的时候，仍然可以通过快照及时恢复数据，你也可以复制一个数据库到同一个区域里或者其他的区域。RDS提供了手动快照和自动快照的功能，并且可以对RDS数据库实例进行基于时间点的恢复。

在本节中，读者将了解如何使用RDS快照：

- 为自动快照配置保留期限和时间窗口；
- 手动创建快照；
- 从创建的快照恢复数据库到一个新的数据库实例；
- 复制快照到其他的区域，以进行跨区域容灾或者数据迁移。

### 9.3.1 配置自动快照

在9.1节中创建的WordPress博客的RDS数据库可以自动为数据库创建快照。在每天特定的时间段，RDS会为数据库创建全自动的快照。如果没有指定特定的时间窗口，RDS会在晚上随机选择一个30 min的时间窗口来创建快照。默认情况下，全自动快照在一天后会被删除；可以修改保留期为1~35天的任意时间段。

创建快照需要暂停所有磁盘的操作。对数据库的访问请求可能会被推迟响应，甚至在超时后失败，所以推荐选择一个对应和用户影响最小的时间段进行全自动快照的操作。

下面的命令将把默认的自动快照时间窗口改为UTC时间05:00~06:00，保留期改为3天。在本机的终端上执行下面的命令：

```
$ aws cloudformation update-stack --stack-name wordpress --template-url \
https://s3.amazonaws.com/awsinaction/chapter9/template-snapshot.json \
--parameters ParameterKey=KeyName,UsePreviousValue=true \
ParameterKey=AdminPassword,UsePreviousValue=true \
ParameterKey=AdminEmail,UsePreviousValue=true
```

RDS数据库将根据修改后的CloudFormation模板做修改，如代码清单9-4所示。

代码清单9-4 修改RDS数据库的快照时间窗口和保留期

```
[...]
"Database": {
  "Type": "AWS::RDS::DBInstance",
  "Properties": {
    "AllocatedStorage": "5",
    "DBInstanceClass": "db.t2.micro",
    "DBInstanceIdentifier": "awsinaction-db",
    "DBName": "wordpress",
    "Engine": "MySQL",
    "MasterUsername": "wordpress",
    "MasterUserPassword": "wordpress",
    "VPCSecurityGroups": [
      {"Fn::GetAtt": ["DatabaseSecurityGroup", "GroupId"]}
    ],
    "DBSubnetGroupName": {"Ref": "DBSubnetGroup"},
    "BackupRetentionPeriod": 3,      <-- 保留快照3 天
    "PreferredBackupWindow": "05:00-06:00"  <-- 在05:00~06:00（UTC时间）
    自动创建快照
  }
}
[...]
```

如果用户想要禁用自动快照，可以修改保留期为0。通常可以使用CloudFormation模板，管理控制台或者SDK来配置全自动快照。

## 9.3.2 手动创建快照

在全自动快照之外，用户还可以在需要的时候手动创建快照。下面的命令将创建一个名为wordpress-manual-snapshot 的快照：

```
$ aws rds create-db-snapshot --db-snapshot-identifier \
```

```
wordpress-manual-snapshot \  
--db-instance-identifier awsinaction-db
```

创建快照大概需要几分钟的时间。用户可以使用下面的命令检查快照的状态：

```
$ aws rds describe-db-snapshots \  
--db-snapshot-identifier wordpress-manual-snapshot
```

RDS不会自动删除手动创建的快照；如果不再需要它们，用户需要自己手动删除。本节的最后将介绍如何操作。

#### 复制自动快照到手动快照

自动快照和手动快照不一样：自动快照在保留期过后会自动删除，但是手动快照不会。如果希望在保留其后仍然保留自动快照，必须把自动快照复制成手动快照。

在本地终端中输入下面的命令，可以获取在9.1节中创建的RDS数据库的自动快照的快照标ID：

```
$ aws rds describe-db-snapshots --snapshot-type automated \  
--db-instance-identifier awsinaction-db \  
--query DBSnapshots[0].DBSnapshotIdentifier \  
--output text
```

下面的命令把自动快照复制为名为**wordpress-copy-snapshot** 的手动快照。替换**\$SnapshotId** 为上一步命令的输出：

```
$ aws rds copy-db-snapshot --source-db-snapshot-identifier \  
$SnapshotId --target-db-snapshot-identifier \  
wordpress-copy-snapshot
```

自动快照的副本被命名为**wordpress-copy-snapshot**，不会被自动删除。

## 9.3.3 恢复数据库



从自动快照或者手动快照恢复时，将会基于快照创建一个新的数据库。如图9-2所示，你不能把快照恢复到一个已有的数据库。

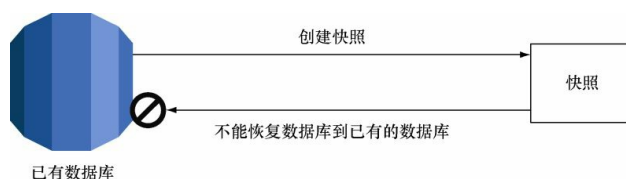


图9-2 不能把快照恢复到已有的数据库

图9-3所示为了恢复快照，创建一个新数据库。

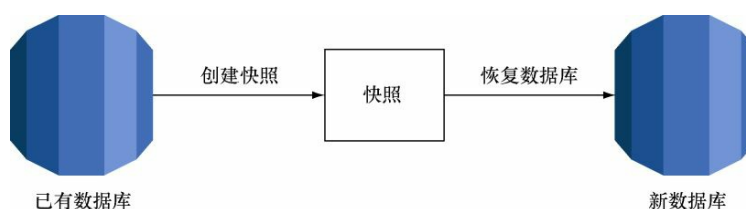


图9-3 为了恢复快照创建一个新数据库

#### 使用DNS CNAME指向用户的数据库

每个RDS数据库会获得一个DNS名称，类似awsinaction-db.czwnecyjmj.us-east-1.rds.amazonaws.com。从快照恢复数据库后，新的数据库实例将会获得一个新的名字。如果把数据库名硬编码到应用的配置中，应用程序将无法工作，因为它没有使用新的DNS名。为了避免这种情况，可以创建一个DNS记录如mydatabase.mycompany.com，通过CNAME指向数据库的DNS名称。在需要恢复数据库时，修改DNS记录指向新的数据库；应用程序就可以重新工作，因为使用mydatabase.mycompany.com的域名连接数据库。AWS的DNS服务是Route 53。

想要在9.1节中创建的VPC里创建一个新数据库，需要找到已有数据库的子网组。获取该信息需要执行下面的命令：

```
$ aws cloudformation describe-stack-resource \
--stack-name wordpress --logical-resource-id DBSubnetGroup \
--query StackResourceDetail.PhysicalResourceId --output text
```

现在可以基于之前创建的手动快照来创建一个新的数据库。替换\$SubnetGroup后执行下面的命令：

```
$ aws rds restore-db-instance-from-db-snapshot \  
--db-instance-identifier awsinaction-db-restore \  
--db-snapshot-identifier wordpress-manual-snapshot \  
--db-subnet-group-name $SubnetGroup
```

基于手动快照创建的新的数据库名为**awsinaction-db-restore**。在数据库创建后，可以切换WordPress应用到新的端点。

使用自动创建的快照，就可以把数据库恢复到一个特定的时间点，因为RDS保存了数据库的变更日志。这样就可以把数据库恢复到从备份保留期开始到最近的5 min的任意一个时间点。

在下面的命令中替换**\$subnetGroup**为之前的**describe-stack-resource** 命令的输出，并且替换**\$Time** 为一个5 min前的时间（如**2015-05-23T12:55:00Z**，UTC时间），然后运行下面的命令：

```
$ aws rds restore-db-instance-to-point-in-time \  
--target-db-instance-identifier awsinaction-db-restore-time \  
--source-db-instance-identifier awsinaction-db \  
--restore-time $Time --db-subnet-group-name $SubnetGroup
```

这样就可以基于5 min前的源数据库创建起来一个新的名为**awsinaction-db-restore-time**的数据库。在数据库创建完成后，可以切换WordPress应用到新的端点。

## 9.3.4 复制数据库到其他的区域

使用快照可以方便地把数据库复制到其他区域。可能基于下面的原因跨区域复制数据库。

- 灾难恢复 ——可以区域基本进行灾难恢复。
- 迁移 ——把基础架构迁移到另外一个区域，这样改善用户的访问延时。

你可以轻松地把快照复制到其他区域。下面的命令把名为 `wordpress-manual-snapshot` 的快照从 `us-east-1` 区域复制到 `eu-west-1`。在执行命令之前需要替换 `$AccountId`：

```
$ aws rds copy-db-snapshot --source-db-snapshot-identifier \
arn:aws:rds:us-east-1:$AccountId:snapshot:\
wordpress-manual-snapshot --target-db-snapshot-identifier \
wordpress-manual-snapshot --region eu-west-1
```

#### 注意

跨区域移动数据可能违反隐私法律或者法规规定，特别是数据跨越国界的时候。在跨区域移动真实的数据前确保你被允许这么做。

如果用户记不得自己的账户ID，可以使用下面的命令行查看：

```
$ aws iam get-user --query "User.Arn" --output text
![p240a1{44}](/api/storage/getbykey/original?key=1806c00bbca174f6d207)arn:
aws:iam::878533158213:user/mycli
```

快照复制到 `eu-west-1` 区域后，就可以像之前介绍的内容恢复数据库。

## 9.3.5 计算快照的成本

快照基于使用的存储容量收费。用户可以免费存储和自己数据库实例相同容量的快照。在WordPress博客平台的这个例子中，你可以免费存储最多5 GB的快照。超过的部分，按照每GB每月使用的存储容量付费。在编写本书的时候，每GB每月成本为0.095美元（在 `us-east-1` 区域）。

#### 资源清理

现在需要清理创建的数据库和快照。按顺序执行下面的命令：

```
$ aws rds delete-db-instance --db-instance-identifier \
awsinaction-db-restore --skip-final-snapshot      <--删除从快照中恢复的数据库
$ aws rds delete-db-instance --db-instance-identifier \
awsinaction-db-restore-time --skip-final-snapshot  <--删除基于时间点恢复的数据库
$ aws rds delete-db-snapshot --db-snapshot-identifier \
wordpress-manual-snapshot      <--删除手动创建的快照
$ aws rds delete-db-snapshot --db-snapshot-identifier \
wordpress-copy-snapshot        <--删除复制的快照
$ aws --region eu-west-1 rds delete-db-snapshot --db-snapshot-identifier \
wordpress-manual-snapshot      <--删除复制到另一个区域的快照
```

## 9.4 控制对数据库的访问

责任共担的安全模型适用于RDS服务，也适用于其他AWS服务。在本例中AWS为云上的安全负责，如底层操作系统的安全。作为客户，你需要制定规则控制对数据和RDS数据库的访问。

图9-4展示了对于RDS数据库进行访问控制的3个层面。

- 控制对RDS数据库的配置的访问。
- 控制对RDS数据库的网络访问。
- 使用用户和权限管理来控制对数据库自身的数据访问控制。

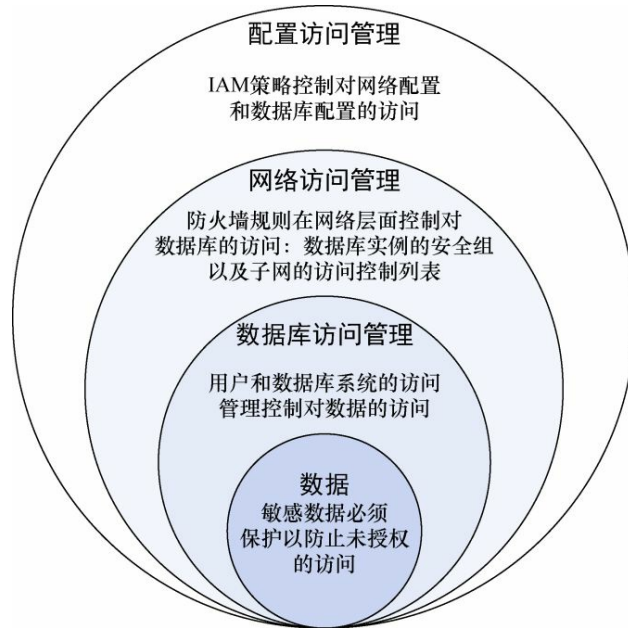


图9-4 数据访问通过数据库访问管理，安全组 and IAM 来控制

### 9.4.1 控制对RDS数据库的配置的访问控制

身份和访问控制管理服务（IAM）可以帮助控制对RDS服务的访问。IAM服务负责控制诸如对创建、更新和删除RDS数据库实例等操作的访问。IAM不管理数据库内部的访问；数据库引擎负责那部分的安全

控制（见9.4.3节）。IAM的策略定义了一个用户或者用户组允许执行的RDS服务的配置和管理操作。把IAM策略关联到特定的IAM用户，用户组或者角色，被关联的实体就可以使用该策略配置RDS数据库。

代码清单9-5显示的IAM策略允许对RDS服务进行所有配置和管理操作。仅把策略关联到特定的IAM用户和组以限制访问。

代码清单9-5 IAM策略允许管理RDS的权限

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "Stmt1433661637000",
    "Effect": "Allow",          <-- IAM策略允许对特定自由的特定操作
    "Action": ["rds:*"],       <-- 允许对RDS 服务的所有可能操作（例如，修改数据库
                                的配置）
    "Resource": "*"           <-- 指定了所有的RDS 数据库资源
  }]
}
```

应该仅授权给真正需要更改RDS数据库的人或者服务器。如果对IAM服务感兴趣，可以查看第6章的内容。

## 9.4.2 控制对RDS数据库的网络访问

RDS实例关联到安全组。安全组包含了一组防火墙规则，控制数据库入站和出站的流量。你已经了解过如何把安全组配合虚拟服务器来工作。

代码清单9-6展示了在WordPress示例中创建的RDS数据库所使用的安全组。这里的规则仅仅允许来源为WebServerSecurityGroup 的网络流量对3306端口的入站访问（3306为MySQL的默认端口）。

代码清单9-6 CloudFormation模板片段：RDS数据库的防火墙规则

```
{
  [...]
```

```

"Resources": {
  [...]
  "DatabaseSecurityGroup": {      <--数据库实例的安全组，允许Web服务器访问MySQL 的默认端口
    "Type": "AWS::EC2::SecurityGroup",
    "Properties": {
      "GroupDescription": "awsinaction-db-sg",
      "VpcId": {"Ref": "VPC"},
      "SecurityGroupIngress": [{
        "IpProtocol": "tcp",
        "FromPort": "3306",      <--MySQL 默认端口为3306
        "ToPort": "3306",
        "SourceSecurityGroupId": {"Ref": "WebServerSecurityGroup"}      <--来源为Web 服务器所在的安全组
      }]
    }
  },
  [...]
},
[...]
```

在网络层面上，应该仅允许真正需要连接到RDS数据库的服务器的入站访问。如果你有兴趣，请到第6章查看安全组和防火墙规则的详细信息。

### 9.4.3 控制数据访问

数据库引擎提供了访问权限控制。数据库引擎的用户管理和IAM用户的权限没有任何关系；它只用来控制对数据库的访问。例如，通常需要为每个应用程序创建一个用户，并且在必要的时候为该用户分配访问和操作表的权限。

常见的使用场景如下：

- 限制特定的用户对数据库的写操作（例如，仅授权给应用程序）。
- 仅允许特定的用户访问特定的表（例如，授权给某个组织的某个部门）。
- 通过对表的访问限制来隔离不同的应用程序（例如，允许不同的客

户的多个应用程序的访问同一个数据库)。

不同的数据库系统使用的用户和权限管理也不尽相同。本书中不会介绍这部分内容；请参考数据库的文档来了解相关信息。



## 9.5 可以依赖的高可用的数据库

数据库是一个系统中最重要的一部分。如果数据库无法访问，应用程序就无法正常工作，存储在数据库里的数据是业务关键型数据，所以数据库必须是高可用并且持久化地存储数据。

Amazon RDS让你运行一个高可用的数据库。和默认的包含一个实例的数据库相比，高可用的RDS数据库包含两个数据库实例：一个主库和一个从库。如果运行一个高可用的RDS数据库，你需要为两个实例付费。所有的客户端请求发送到主库。就像图9-5显示的那样，数据在主库和从库之间同步复制。

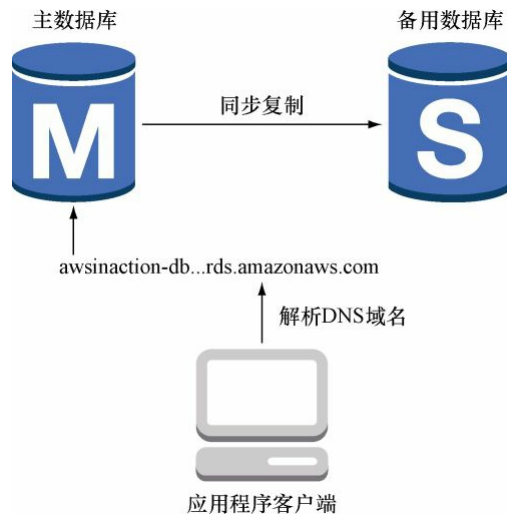


图9-5 运行在高可用模式时主数据库复制到从数据库

如果主库因为硬件或者网络故障，RDS会启动故障切换流程。从库提升为主库。如图9-6所示，DNS名称会被更新，客户端的请求将发送给之前的从库。

RDS自动侦测故障并进行切换，不需要人工干预。对于生产负载，强烈推荐使用高可用的部署方式。

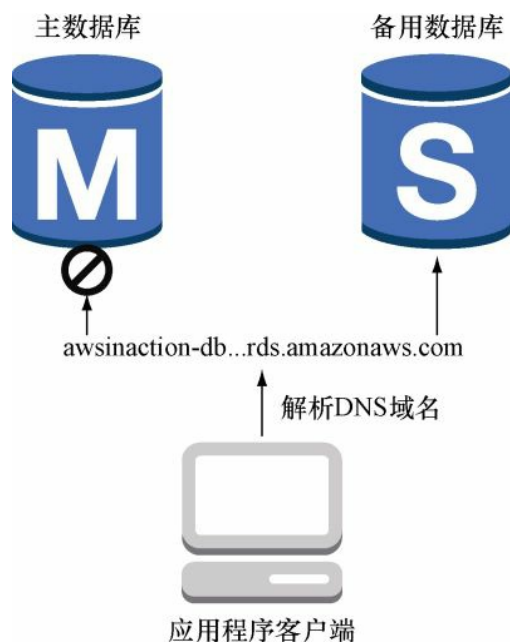


图9-6 主数据库发生故障后客户端程序通过DNS名字解析切换到从数据库

## 激活RDS数据库的高可用部署选项

在本地的终端输入下面的命令来激活RDS数据库的高可用部署，用于WordPress博客平台使用：

```
$ aws cloudformation update-stack --stack-name wordpress --template-url \
https://s3.amazonaws.com/awsinaction/chapter9/template-multiaz.json \
--parameters ParameterKey=KeyName,UsePreviousValue=true \
ParameterKey=AdminPassword,UsePreviousValue=true \
ParameterKey=AdminEmail,UsePreviousValue=true
```

### 警告

启动高可用部署的RDS数据库将产生费用。如果想了解当前的价格信息，可以访问AWS官方网站。

我们使用稍修改过的CloudFormation模板来更新RDS数据库，如代码清单9-7所示。

```
[...]
"Database": {
  "Type": "AWS::RDS::DBInstance",
  "Properties": {
    "AllocatedStorage": "5",
    "DBInstanceClass": "db.t2.micro",
    "DBInstanceIdentifier": "awsinaction-db",
    "DBName": "wordpress",
    "Engine": "MySQL",
    "MasterUsername": "wordpress",
    "MasterUserPassword": "wordpress",
    "VPCSecurityGroups": [
      {"Fn::GetAtt": ["DatabaseSecurityGroup", "GroupId"]}
    ],
    "DBSubnetGroupName": {"Ref": "DBSubnetGroup"},
    "MultiAZ": true      <-- 为RDS 数据库激活高可用部署
  }
}
[...]
```

数据库大概需要几分钟的时间才能进入高可用模式。但是不需要做任何其他事情——现在数据库已经支持高可用了。

#### 什么是多可用区部署

每个AWS区域都包含多个独立的数据中心，也被称为可用区。第11章将介绍可用区的概念。所以这里暂时略过对RDS跨可用区部署（把RDS的主库和从库启动在两个不同的可用区）的介绍。AWS把这种高可用的部署方式称为RDS多可用区部署。

使用RDS高可用的部署除了提高数据库可靠性之外，还有其他的好处。重新配置或者维护数据库一般将导致停机。一个高可用部署的RDS数据库允许在维护的时候切换到从库，从而解决了这个问题。

## 9.6 调整数据库的性能

通常情况下，RDS数据库，或者任何SQL数据库，可以在垂直方向上扩展。如果数据库性能不足，就必须增加底层硬件的性能：

- 更快的CPU；
- 更多的内存；
- 更高性能的I/O存储。

和关系数据库不同的是，S3这样的对象存储或者DynamoDB这样的NoSQL数据库可以在水平方向上进行扩展。可以通过向集群中添加节点的方式增加性能。

### 9.6.1 增加数据库资源

在启动RDS数据库的时候，需要选择一种实例类型。实例类型决定了虚拟服务器的计算能力和内存容量（和启动一台EC2实例一样）。选择更大的实例类型能增加数据库可以使用的处理性能和内存容量。

这里启动的是db.t2.micro类型的RDS数据库，这是最小的实例类型。可以通过CloudFormation模板、命令行、软件开发工具包SDK和管理控制台来修改实例类型。代码清单9-8显示如何使用CloudFormation模板来把拥有1个虚拟内核和615 MB内存的db.t2.micro修改为2倍虚拟内核和7.5 GB内存的db.m3.large实例。这里只是在理论上介绍如何操作——不要去扩展你正在运行的数据库。

代码清单9-8 修改实例类型来改善RDS数据库的性能

```
{
  [...]
  "Resources": {
    [...]
    "Database": {
      "Type": "AWS::RDS::DBInstance",
      "Properties": {
        "AllocatedStorage": "5",
```

```

        "DBInstanceClass": "db.m3.large",      <-- 把数据库实例底层的虚拟服务器
类型从db.t2.micro 修改为db.m3.large
        "DBInstanceIdentifier": "awsinaction-db",
        "DBName": "wordpress",
        "Engine": "MySQL",
        "MasterUsername": "wordpress",
        "MasterUserPassword": "wordpress",
        "VPCSecurityGroups": [
            {"Fn::GetAtt": ["DatabaseSecurityGroup", "GroupId"]}
        ],
        "DBSubnetGroupName": {"Ref": "DBSubnetGroup"}
    }
},
[...],
},
[...],
}

```

因为数据库必须从磁盘读取和写入数据，所以I/O性能对数据库的总体性能来说至关重要。RDS提供3种不同的存储，你已经在EBS的介绍中了解到了这些EBS存储类型：

- 通用SSD磁盘；
- 预配置IOPS性能的SSD磁盘；
- 物理磁盘。

针对生产系统的工作负载，应该选择通用SSD磁盘或者选择预配置IOPS性能的SSD磁盘。这些选项和你为虚拟服务器选择的EBS存储服务选项一样。如果希望保证高水平的读和写吞吐量，应该使用预配置IOPS性能的SSD固态硬盘。通用SSD提供了中等的性能，并且在需要的时候可用突发性能来满足突发的工作负载的需求。通用SSD的基线性能取决于最初配置的存储容量。如果需要以很低的成本存储数据或者不需要保证访问的性能，可以选择物理磁盘。代码清单9-9显示如何使用CloudFormation来激活通用SSD存储。

代码清单9-9 修改RDS数据库的存储类型来改善性能

```

{
    [...]
    "Resources": {
        [...]
    }
}

```

```

"Database": {
  "Type": "AWS::RDS::DBInstance",
  "Properties": {
    "AllocatedStorage": "5",
    "DBInstanceClass": "db.t2.micro",
    "DBInstanceIdentifier": "awsinaction-db",
    "DBName": "wordpress",
    "Engine": "MySQL",
    "MasterUsername": "wordpress",
    "MasterUserPassword": "wordpress",
    "VPCSecurityGroups": [
      {"Fn::GetAtt": ["DatabaseSecurityGroup", "GroupId"]}
    ],
    "DBSubnetGroupName": {"Ref": "DBSubnetGroup"},
    "StorageType": "gp2"      <--使用通用类型（SSD）存储来改善I/O 性能
  }
},
[...],
},
[...],
}

```

## 9.6.2 使用读副本来增加读性能

SQL数据库可以在特定的情况下水平扩展性能。服务大量读请求的数据库可以通过水平添加更多的读副本的数据库实例的方式来扩展性能。如图9-7所示，数据库的修改以异步的方式复制到额外的只读数据库实例。可以在主数据库和读副本之间分担读请求，来增加读吞吐量。

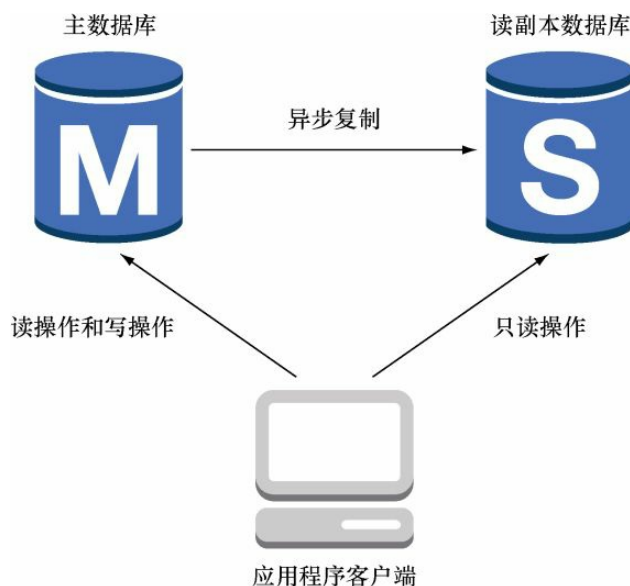


图9-7 在主数据库和读副本数据库之间分担读负载，以提升读性能

通过读副本的方式改善性能只适用于大量读操作和少量写操作的应用类型。幸运的是，大多数应用读取的操作远远大于写入操作。

## 1. 创建一个读副本的数据库

Amazon RDS支持创建MySQL和PostgreSQL数据库的只读副本。要使用读复制，需要激活数据库的自动备份，将在本章的最后一节显示如何操作。

从本地终端执行下面的命令，为在9.1节中的WordPress博客平台的数据库创建只读副本：

```
$ aws rds create-db-instance-read-replica \  
--db-instance-identifier awsinaction-db-read \  
--source-db-instance-identifier awsinaction-db
```

RDS在后台自动触发下列操作。

- (1) 从源数据库（也叫做主数据库）创建快照。
- (2) 从该快照创建新的数据库。

(3) 激活主数据库和只读数据库的复制关系。

(4) 为只读副本数据库创建SQL只读请求的端点。

一旦只读数据库成功创建，它就可以开始接收SQL读请求。应用程序需要支持使用只读副本的数据库。例如，WordPress默认不支持使用只读副本数据库，但是可以通过安装一个叫作HyperDB的插件来支持；它的配置有些复杂，这里略过这部分内容。创建或者删除一个只读副本不会影响主库的可用性。

#### 使用只读副本来跨区域的传输数据

RDS支持MySQL数据库的跨区域复制。例如，可以把位于弗吉尼亚的数据中心里的数据库复制到爱尔兰的数据中心。这个功能主要有下面的使用场景。

- (1) 跨区域的备份数据，以防范极少出现的区域级别的故障。
- (2) 跨区域迁移数据，以满足本地用户读请求所需要的低延时。
- (3) 跨区域迁移数据库。

跨区域的复制数据库会产生额外的成本，因为你还需要为传输的数据流量付费。

## 2. 提升读副本数据库为单独的数据库

如果需要跨区域迁移数据库，或者需要为主数据库分担像添加索引这样高负载的任务，把这些负载从主库切换到只读副本会很有帮助。只读副本必须成为新的主数据库。RDS的MySQL和PostgreSQL的只读数据库可以提升为主数据库。

下面的命令把创建的只读副本数据库提升为单独的主数据库。注意只读数据库将会重新启动并且在几分钟内不可用：

```
$ aws rds promote-read-replica --db-instance-identifier awsinaction-db-read
```

名为awsinaction-db-read的RDS数据库从只读从库提升为主数据库之后，就可以开始接受写请求。



#### 资源清理

现在应该清理不再需要的资源。执行下面的命令：

```
$ aws rds delete-db-instance --db-instance-identifier \
awsinaction-db-read --skip-final-snapshot
```

在本章我们已经积累了一定的AWS关系数据库服务的经验。最后我们来了解如何紧密监控RDS的性能。

## 9.7 监控数据库

RDS是一个托管的服务。但是，用户仍然需要监控一些指标，以确保数据库可以响应所有来自应用程序的访问请求。RDS发布一些指标给AWS CloudWatch服务，CloudWatch是一个AWS云的监控服务。像图9-8显示的那样，可以通过管理控制台查看这些指标，并且定义超过一定阈值之后产生报警。

访问RDS数据库的指标可以按照下面的步骤。

- (1) 打开管理控制台。
- (2) 在主菜单中选择CloudWatch服务。
- (3) 在左侧选择RDS指标的子菜单。
- (4) 在显示的表下选择想要查看的指标。

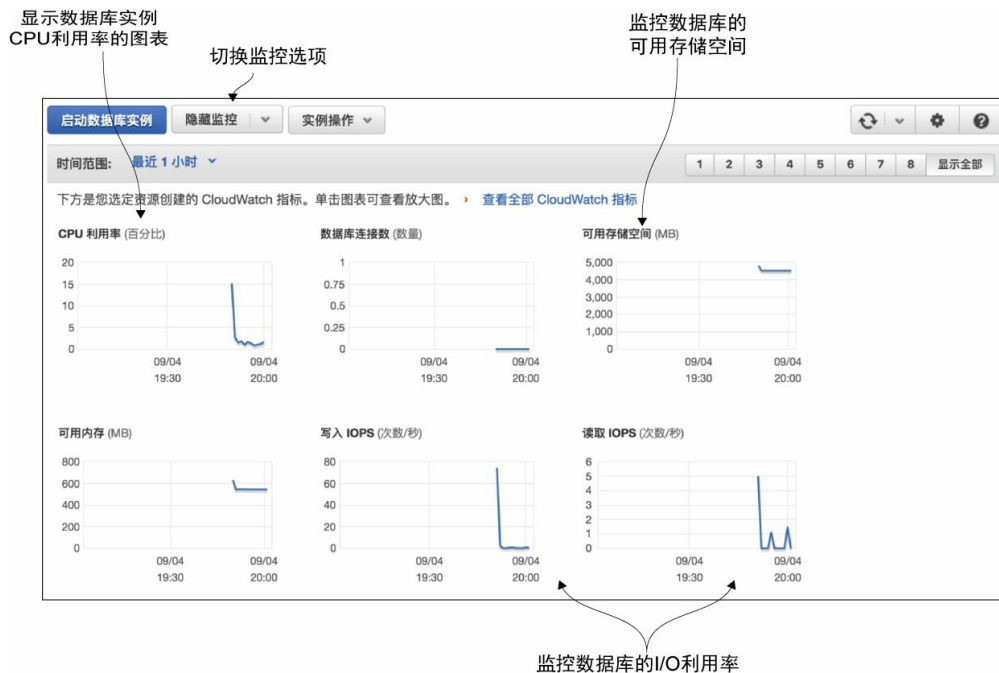


图9-8 从管理控制台监控RDS数据库的指标

RDS数据库实例提供18个指标，表9-5展示了重要的指标；推荐通

过创建报警的方式持续监控这些指标。

表9-5 从CloudWatch监控RDS数据库的重要指标

名 称	描 述
可用存储空间	可用的存储容量，字节为单位。确保没有用完存储容量
CPU利用率	CPU利用率以百分比显示。高利率用率可能意味着CPU性能瓶颈
可用内存	可用内存容量，字节为单位。内存不足会导致性能问题
磁盘队列深度	磁盘的等待请求数量。一个长队列意味着数据库遇到了存储I/O性能瓶颈

推荐对这些指标给予特别的关注，以确保数据库不会为应用程序带来性能问题。

资源清理

现在清理资源来避免不必要的花费。执行下面的命令来删除为WordPress博客平台创建的RDS数据库相关资源：

```
$ aws cloudformation delete-stack --stack-name wordpress
```

本章中我们学习了如何使用RDS服务来管理应用的关系数据库。下一章我们将专门讨论一个NoSQL数据库。

## 9.8 小结

- RDS是一个提供关系数据库的托管服务。
- 可以选择MySQL、PostgreSQL、Microsoft SQL、Oracle、MariaDB和Amazon Aurora这样的数据库引擎。
- 最方便地把数据导入RDS数据库的方法，是复制数据到同一个区域的一台虚拟服务器，然后从该虚拟服务器导入数据到RDS数据库。
- 可以配合使用IAM策略和防火墙规则，以及数据库级别的安全工具来控制对数据的访问。
- 在数据保留期内，可以把RDS数据库恢复到任何时间点。
- RDS数据库可以是高度可用的。对于生产负载，应该以多AZ模式启动RDS数据库。
- 读副本数据库可以改善SQL数据库的读密集应用的性能。

## 第10章 面向NoSQL数据库服务的编程： DynamoDB

### 本章主要内容

- DynamoDB的NoSQL数据库服务
- 创建表和二级索引
- 在服务堆栈里集成DynamoDB
- 设计键值优化的数据模型
- 优化性能

扩展传统的关系数据库的性能非常困难，因为保证事务（原子性、一致性、隔离、持久性，也叫作ACID）需要跨数据库的所有节点通信。添加的节点越多，数据库会变得越慢，因为更多的节点需要彼此协调交易操作。要应对这样的难题需要使用不提供上面ACID保证的数据库类型。它们叫作NoSQL数据库。

有4种主要的NoSQL数据库（文档、图形、列式和键值存储）每种类型有适用的场景和应用程序。亚马逊提供一个叫作DynamoDB的NoSQL数据库服务。和RDS不一样，DynamoDB是完全托管的，非开源的键值存储。如果希望使用不同的NoSQL数据库类型（如MongoDB这样的文档数据库），你需要启动EC2实例并且在上面自己安装管理MongoDB。第3章和第4章的会介绍如何操作。DynamoDB高可用和高度持久的。你可以把DynamoDB的容量从一个项目扩展到存储几十亿个项目，也可以把它的性能从每秒钟一个操作扩展到每秒上万个操作。

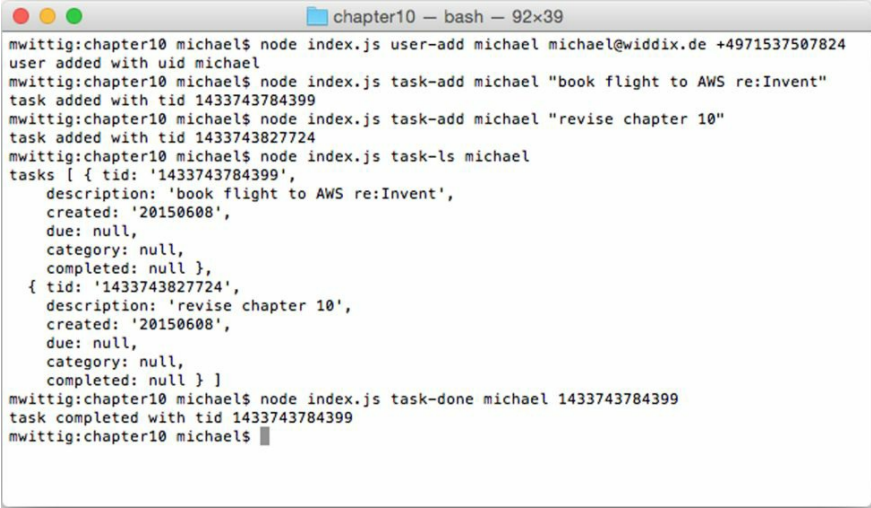
本章详细了解如何使用DynamoDB：如何管理它和如何编程以使用DynamoDB。管理DynamoDB非常简单。可以为DynamoDB创建表和二级索引。性能相关的可以调整的参数只有一个：读容量和写容量单位，读写容量单位直接决定了表的性能和成本。

我们将查看DynamoDB的细节，并通过一个简单的任务管理应用nodetodo（类似Hello World程序）来展示如何使用它。图10-1展示了如

何使用nodetodo的任务管理应用。

示例都包含在免费套餐中

本章中的所有示例都包含在免费套餐中。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

A terminal window titled 'chapter10 - bash - 92x39' showing a series of commands and their outputs for the 'nodetodo' application. The commands include adding a user, adding tasks, listing tasks, and marking a task as done. The output shows the user 'michael' being added with a specific UID, two tasks being added with their respective TIDs, the tasks being listed as a JSON array, and one task being marked as completed.

```
mwittig:chapter10 michael$ node index.js user-add michael michael@widdix.de +4971537507824
user added with uid michael
mwittig:chapter10 michael$ node index.js task-add michael "book flight to AWS re:Invent"
task added with tid 1433743784399
mwittig:chapter10 michael$ node index.js task-add michael "revise chapter 10"
task added with tid 1433743827724
mwittig:chapter10 michael$ node index.js task-ls michael
tasks [ { tid: '1433743784399',
  description: 'book flight to AWS re:Invent',
  created: '20150608',
  due: null,
  category: null,
  completed: null },
  { tid: '1433743827724',
  description: 'revise chapter 10',
  created: '20150608',
  due: null,
  category: null,
  completed: null } ]
mwittig:chapter10 michael$ node index.js task-done michael 1433743784399
task completed with tid 1433743784399
mwittig:chapter10 michael$
```

图10-1 你可以使用nodetodo应用的命令行接口管理你的任务

在开始实现nodetodo之前，需要了解DynamoDB的最基本的信息。

## 10.1 操作DynamoDB

DynamoDB不需要任何传统关系数据库的管理操作，但是有其他需要关注的任务。DynamoDB的价格取决于存储容量和性能需求。本节还比较了DynamoDB和RDS的不同。

### 10.1.1 管理

有了DynamoDB，你不需要担心安装、更新、服务器、存储或者备份操作。

- DynamoDB不是一个可以下载的软件产品。相反的，它是一个NoSQL的数据库服务。所以，不能像安装一个MySQL或者MongoDB一样安装DynamoDB。这也意味着不能为数据库打补丁；DynamoDB的软件是由AWS来维护的。
- DynamoDB运行在AWS运维管理的一批服务器上。他们负责操作系统和安全相关的问题。从安全的角度来说，你的任务是分配合适的权限给IAM用户访问DynamoDB表。
- DynamoDB跨服务器和多个数据中心复制数据。没有必要出于持久化保存数据的目的来做备份——数据库已经帮助在物理层面保护了数据。

现在了解到了使用DynamoDB，用户不再需要进行某些管理操作。但是，要在生产环境里使用DynamoDB，仍然需要考虑一些事情：创建表（见10.4节）、创建二级索引（见10.6节）、监控容量使用和配置读写容量（见10.9节）。

### 10.1.2 价格

如果使用DynamoDB，每月需要为下面的用量付费。

- 每GB每月容量0.25美元（二级索引通用消耗容量）。

- 每10个写容量单位0.47美元（10.9节将解释读写容量单位）。
- 每50个读容量单位0.09美元。

这些价格信息适用于当前的弗吉尼亚北部的区域（us-east-1）。如果使用在同一区域的EC2服务器访问DynamoDB，不会产生额外的流量费。

### 10.1.3 与RDS对比

表10-1比较了DynamoDB和RDS。注意这就像拿着苹果和橘子对比，并不是对等的比较。DynamoDB和RDS唯一的共同点就是它们都叫作数据库。

表10-1 DynamoDB和RDS的区别

任 务	DynamoDB	RDS
创建一个表	管理控制台，SDK或者CLI <code>aws dynamodb create-table</code>	SQL CREATE TABLE 语句
插入、更新或者删除数据	SDK	SQL分别使用INSERT、UPDATE或DELETE 语句
查询数据	如果查询主键：SDK 无法查询非主键属性，但是可以添加二级索引或者扫描整张表	SQL SELECT 语句
增加存储	无须任何操作；DynamoDB随着项目的增加自动扩容	调配更多存储
增加性能	水平方向，通过增加读写容量单位。DynamoDB会在底层增加服务器	垂直方向，通过增加实例大小；或者水平，通过增加只读副本。有容量上限



在本地安装数据库	DynamoDB不提供下载。仅可以以服务的方式使用	下载数据库软件并安装在本地
雇佣专家	寻找掌握DynamoDB的人才	寻找熟悉SQL或者专业技术的人才，取决于具体的数据库引擎

## 10.2 开发者需要了解的DynamoDB内容

DynamoDB以键值存储的方式组织表里的数据。每个表包含了使用键（Key）来代表的项目（值）。一个表还可以包含二级索引，用来提供基于主键以外的数据查询功能。在本节中，你将看到这些DynamoDB的基本组件，并且最后将比较它和其他NoSQL数据库的不同。

### 10.2.1 表、项目和属性

DynamoDB的表需要起一个名字，表里组织项目的集合。一个项目包含了属性的集合。一个属性就是一个名—值对。属性的值可以是基本类型（如数值、字符串、二进制、布尔型）、复合类型（数字集、字符串集、二进制集）或者一个JSON文档（对象、数组）。在表里的项目不需要一定有相同的属性；没有强制的模式。

可以从管理控制台、CloudFormation、SDK或者命令行来创建表（现在先不要使用下面的命令创建表，本章稍后将创建一个DynamoDB表）：

```
$ aws dynamodb create-table --table-name app-entity \      <---为表选择一个名字，如app-entity
--attribute-definitions AttributeName=id,AttributeType=S \      <---名为id的属性类型为字符串
--key-schema AttributeName=id,KeyType=HASH \      <---主键使用id 属性
--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5      <---
在10.9 节将提供读容量单位和写容量单位的内容
```

如果计划让多个应用程序使用DynamoDB，好的实践是用应用程序的名称作为表名字的前缀。可以通过管理控制台创建表。注意不能修改表的名字，也不能修改主键的模式。但是随时可以添加属性的定义和修改配置的吞吐量。

## 10.2.2 主键

主键在表里唯一，并唯一地标识一个项目。你需要使用主键来查找一个项目。主键可以是一个分区键或者由一个分区键和一个排序键组成复合主键。

### 1. 分区键

一个分区键使用项目的一个属性来创建散列索引。如果希望基于分区键查找一个项目，需要知道准确的分区键值。一个用户表可以使用用户电子邮件作为分区主键。如果你知道用户的分区键（本例中是电子邮件地址），就可以获取对应的项目的数据。

### 2. 分区键和排序键

分区键和排序键使用项目的两个属性来创建更加强大的索引功能。第一个属性是键的分区键，第二个属性是排序键。要查找一个项目，仍然需要知道项目的分区键，但是不需要知道它的排序键。相同分区键不同排序键的项目顺序存储在同一个分区。这样可以从一个特定的出发点进行范围查询。一个消息表可以使用分区和排序键作为主键；分区键是用户的电子邮件，排序键是时间戳。你可以查找用户在特定时间之后发的所有消息。

## 10.2.3 与其他NoSQL数据库的对比

表10-2对DynamoDB和其他NoSQL数据库做了比较。但要记住，所有数据库都有优势和劣势，表10-2只展示了高级别的比较和如何在AWS上使用。

表10-2 DynamoDB和一些NoSQL数据库的不同

任 务	DynamoDB 键值存储	MongoDB 文 档存储	Neoj 图形存 储	Cassandra 列式存储	Riak KV 键 值存储
-----	------------------	------------------	---------------	-------------------	------------------

在AWS运行生产数据库	一键部署；它是托管服务	EC2实例集群，自己运维管理	EC2实例集群，自己运维管理	EC2实例集群，自己运维管理	EC2实例集群，自己运维管理
联机增加可用存储容量	不需要。数据库自动扩容	增加更多的EC2实例（副本集）	不可以（增加EBS卷需要停机）	增加更多的EC2实例	增加更多的EC2实例

## 10.2.4 DynamoDB本地版

假设有一组开发者在使用DynamoDB开发一个新的应用。在开发时，每个开发者需要一个隔离的数据库，以免损坏其他团队成员的数据。他们还需要进行单元测试，以保证他们的应用程序可以正常工作。你可以使用CloudFormation创建一个堆栈来给每个开发人员提供单独的DynamoDB表，或者可以使用DynamoDB本地版。AWS提供一个基于Java的DynamoDB的模型，可以在AWS官方网站下载。别在生产环境里使用它！它仅用于开发环境并且提供和DynamoDB相同的功能，但是底层实现不同：只有API是一样的。

## 10.3 编写任务管理应用程序

为了最小化引入编程语言的复杂性，你将使用Node.js/JavaScript来创建一个小的任务管理应用，可以运行在本地电脑的终端。我们把这个应用称为nodetodo。nodetodo将使用DynamoDB作为数据库。nodetodo可以帮你实现下面的功能：

- 创建和删除用户；
- 创建和删除用户；
- 将任务标记为完成；
- 使用各种过滤条件显示任务列表。

nodetodo支持多个用户，并且可以跟踪有或者没有到期日期的任务。为了帮助用户处理多个任务，任务可以被加入目录。nodetod通过终端访问。下面显示了如何通过终端访问nodetodo并添加一个用户（现在先别运行这个命令，我们还没有实现这个应用）：

```
# node index.js user-add <uid> <email> <phone>      <--CLI 命令行的解释：参数用<>标记
$ node index.js user-add michael michael@widdix.de 0123456789      <--在终端执行nodetodo
=> user added with uid michael      <--nodetodo 的输出写入到STDOUT 标准输出
```

要添加一个新的任务，需要进行如下操作（现在先不要运行这个命令，我们还没有实现这个应用）：

```
# node index.js task-add <uid> <description> \
[<category>] [--dueat=<yyyymmdd>]      <--可选参数用[]标记
$ node index.js task-add michael "plan lunch" --dueat=20150522      <--命名的参数调用，使用--name=value指定参数的值
=> task added with tid 1432187491647      <--tid 是任务的ID
```

用户可以像下面这样将任务标记为完成（现在先不要运行这个命令，我们还没有实现这个应用）：

```
# node index.js task-done <uid> <tid>
$ node index.js task-done michael 1432187491647
=> task completed with tid 1432187491647
```

用户还可以列表查看所有任务。下面的命令显示如何使用`nodetodo`列出任务（现在先不要运行这个命令，我们还没有实现这个应用）：

```
# node index.js task-ls <uid> [<category>] [--overdue|--due|...]
$ node index.js task-ls michael
=> tasks [...]
```

为了实现一个直观的命令行工具，`nodetodo`使用了`docopt`，一个命令行接口描述语言，来描述命令行接口。支持的命令如下所示。

- `user-add` —— 添加新用户到`nodetodo`。
- `user-rm` —— 删除用户。
- `user-ls` —— 用户清单。
- `user` —— 显示一个用户的详细信息。
- `task-add` —— 添加任务到`nodetodo`。
- `task-rm` —— 删除任务。
- `task-ls` —— 使用过滤条件列出用户任务。
- `task-la` —— 使用过滤条件按照目录列出任务。
- `task-done` —— 标记某个任务为完成。

在本章下面的几节，我们将实现这些命令。代码清单10-1展示了所有的CLI的命令描述，包括参数。

代码清单10-1 使用`docopt`的CLI命令行描述：使用`nodetodo`（cli.txt）

```
nodetodo

Usage:
  nodetodo user-add <uid> <email> <phone>
  nodetodo user-rm <uid>
  nodetodo user-ls [--limit=<limit>] [--next=<id>]    <--命名的参数限制和可选的下一个参数
  nodetodo user <uid>
```

```
nodetodo task-add <uid> <description> \  
[<category>] [--dueat=<yyyymmdd>]      <--category 参数是可选的  
nodetodo task-rm <uid> <tid>  
nodetodo task-ls <uid> [<category>] \  
[--overdue|--due|--withoutdue|--futuredue]  <--管道符代表“或者”  
nodetodo task-la <category> \  
[--overdue|--due|--withoutdue|--futuredue]  
nodetodo task-done <uid> <tid>  
nodetodo -h | --help      <--help 列出如何使用nodetodo 的帮助信息  
nodetodo --version        <--版本信息
```

Options:

-h --help	Show this screen.
--version	Show version.

DynamoDB和在关系数据库中使用的创建、读取、更新和删除数据的SQL命令完全不同。你将使用SDK开发工具包调用HTTP REST API来访问DynamoDB。你必须把DynamoDB集成到应用程序里；不能使用一个SQL数据库的应用直接运行访问DynamoDB。要想使用DynamoDB，需要编写代码！

## 10.4 创建表

DynamoDB表组织你的数据。在建表的时候不需要定义表里的项目所有需要的属性。DynamoDB不要求一个静态的模型，这一点和传统数据库不同，但是必须定义要用来做主键的属性。为了做到这一点，我们将使用AWS命令行工具。`aws dynamodb create-table` 命令有以下4个必需的选项。

- **table-name** ——表的名字（不能修改）。
- **attribute-definitions** ——用作主键的属性的名字和类型。可以多次使用`AttributeName=attr1, AttributeType=S`定义，用空格键隔开。合格的数据类型包括S（字符串）、N（数值）和B（二进制）。
- **Key-schema** ——用作主键的一部分的属性的名称（不能修改），包含一个或者两个`AttributeName=attr1, KeyType=HASH` 条目来定义分区键和排序键。合格的类型为HASH 类型和RANGE 类型。
- **provisioned-throughput** ——该表的性能设置，使用`ReadCapacityUnits=5, WriteCapacityUnits=5` 来定义（见10.9节对这部分内容的介绍）。

现在我们要为nodetodo应用程序创建一个用户表，并为所有任务创建一个任务表。

### 10.4.1 使用分区键的用户表

在为nodetodo用户创建表之前，用户必须认真考虑表的名字和主键。我们推荐你使用应用名称作为表名字的前缀。在本例中，表的名字为**todo-user**。在选择主键的时候，你必须考虑将来要运行的查询和数据项目有哪些唯一的数据。用户会有一个唯一的ID，称为**uid**，所以使用**uid**作为主键属性。你必须能够在**user** 命令中使用**uid**来查询用户。如果想使用单一属性作为主键，可以创建一个散列索引：一个基于分区键的没有排序的索引。下面的示例展示了一个用户表，使用**uid**作为主分区键：

---



```

"michael" => {      <---uid ("Michael") 是主分区键; {}包含的所有内容组成一个项目
  "uid": "michael",
  "email": "michael@widdix.de",
  "phone": "0123456789"
}
"andreas" => {      <---分区键没有顺序
  "uid": "andreas",
  "email": "andreas@widdix.de",
  "phone": "0123456789"
}

```

因为用户将仅使用已知的**uid** 来查询，使用分区键是可行的。下面将使用AWS命令行创建一个结构和之前示例相同的用户表：

```

$ aws dynamodb create-table --table-name todo-user \      <---表名使用应用程序
的名字作为前缀，避免未来发生冲突
--attribute-definitions AttributeName=uid,AttributeType=S \      <---项目必须
至少包含一个属性uid，类型为字符串
--key-schema AttributeName=uid,KeyType=HASH \      <---主分区键使用uid 属性
--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5      <---
可以在10.9 节了解此内容

```

创建表需要一定的时间。等到表状态改为**ACTIVE** 即可。可以使用下面的命令检查表的状态：

```

$ aws dynamodb describe-table --table-name todo-user      <---CLI 命令检查表
状态
{
  "Table": {
    "AttributeDefinitions": [      <---表的属性定义
      {
        "AttributeName": "uid",
        "AttributeType": "S"
      }
    ],
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "WriteCapacityUnits": 5,
      "ReadCapacityUnits": 5
    },
    "TableSizeBytes": 0,

```

```

    "TableName": "todo-user",
    "TableStatus": "ACTIVE",      <--- 表的状态
    "KeySchema": [                <--- 定义为主键的属性
      {
        "KeyType": "HASH",
        "AttributeName": "uid"
      }
    ],
    "ItemCount": 0,
    "CreationDateTime": 1432146267.678
  }
}

```

## 10.4.2 使用分区键和排序键的任务表

任务永远属于一个用户，任务相关的所有的命令都包含了用户的ID。为了实现**task-ls**命令，你需要一个访问根据**uid**来查询任务。除了分区键指纹，你可以使用一个分区键和排序键的组合主键。因为和任务相关的所有交互都需要用户ID，你可以选择**uid**作为分区键和一个任务ID（**tid**），创建的时间戳，作为键的排序部分。现在你可以运行包含用户ID的查询，如果需要的话，查询也可以包含任务ID。

### 注意

这个方案有一个限制：用户同一时间戳只能添加一个任务。我们的时间戳使用最小单位为ms（毫秒），这应该没有问题。但是你需要小心防止用户同一时间创建两个任务，这会导致奇怪的结果。

分区键和排序键使用表的两个属性。对于主键的分区部分，系统维护了一个没有排序的散列索引；排序部分存储在一个排序的范围索引。分区和排序键的组合唯一地识别一个项目。下面的数据集展示了没有排序的分区部分和排过顺序的排序键。

```

["michael", 1] => {      <---uid ("Michael") 是主键的分区键，tid (1) 是主键的排序键
  "uid": "michael",
  "tid": 1,

```

```

    "description": "prepare lunch"
  }
  ["michael", 2] => {      <---排序键为使用同一分区键的项目排序
    "uid": "michael",
    "tid": 2,
    "description": "buy nice flowers for mum"
  }
  ["michael", 3] => {
    "uid": "michael",
    "tid": 3,
    "description": "prepare talk for conference"
  }
  ["andreas", 1] => {      <---分区键没有顺序
    "uid": "andreas",
    "tid": 1,
    "description": "prepare customer presentation"
  }
  ["andreas", 2] => {
    "uid": "andreas",
    "tid": 2,
    "description": "plan holidays"
  }
}

```

nodetodo提供获取一个用户的所有任务的功能。如果任务只有一个分区主键，这将非常困难，因为你需要知道所有的主键才能从DynamoDB获取这些数据。幸运的是，分区和排序键让事情更加容易，因为只需要知道主键的分区部分就可以获取所有的项目。对于任务，你将使用**uid** 作为已知分区部分。排序部分是**tid**。任务ID使用创建任务时的时间戳来定义。你将创建任务表，使用两个属性来创建分区和排序索引。

```

$ aws dynamodb create-table --table-name todo-task \
--attribute-definitions AttributeName=uid,AttributeType=S \      <---组合主键
需要至少两个属性
AttributeName=tid,AttributeType=N \
--key-schema AttributeName=uid,KeyType=HASH \
AttributeName=tid,KeyType=RANGE \      <---tid 属性是主键的排序键部分
--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5

```

等待表状态改变，直到aws dynamodb describe-table -

`table-name todo-task` 显示表的状态为**ACTIVE** 。现在两张表都已经就绪，你将添加一些数据。

## 10.5 添加数据

现在有两张表在活跃状态。要使用它们，需要添加一些数据。我们将使用Node.js SDK来访问DynamoDB。在添加用户和任务之前，我们需要先设置SDK和部署现成的代码。

### 安装和开始使用Node.js

Node.js是一个在事件驱动环境下执行JavaScript代码的平台，因此用户可以轻松地构建网络应用。要安装Node.js，可以访问Node.js官方网站并且下载适合的操作系统包。

安装完Node.js，就可以运行`node --version`来确认一切就绪。用户的终端应该返回一些类似`v0.12.*`的输出，此时就可以运行使用AWS的`nodetodo`这样的JavaScript示例。

在开始使用Node.js和`docopt`前，用户需要一些命令来载入所有的依赖包和配置工作。代码清单10-2展示了如何操作。

### 代码在哪里下载

同样，读者可以在下载的源代码中找到这些代码，`nodetodo`位于`chapter10/`。

`docopt`负责读取传递给进程的所有参数。它返回一个JavaScript对象，对象里的变量映射到CLI中描述的参数。

代码清单10-2 `nodetodo`:在Node.js (`index.js`) 中使用`docopt`

```
var fs = require('fs');      <--加载fs 模块访问文件系统
var docopt = require('docopt');  <--加载docopt 模块来读取输入参数
var moment = require('moment');  <--加载moment 模块简化JavaScript中的temporal临时类型
var AWS = require('aws-sdk');    <--从cli.txt 读取CLI的描述
var db = new AWS.DynamoDB({
  "region": "us-east-1"
});

var cli = fs.readFileSync('./cli.txt', {"encoding": "utf8"});  <--
var input = docopt.docopt(cli, {  <--解析参数并保存到输入变量
  "version": "1.0",
```

```
"argv": process.argv.splice(2)
});
```

接下来就实现nodetodo的特性，可以使用putItem SDK操作像下面这样向DynamoDB添加数据：

```
var params = {
  "Item": {      <---所有项目的属性为名称-数值对
    "attr1": {"S": "val1"},      <---字符串类型以S 标记
    "attr2": {"N": "2"}      <---数字类型（浮点型和整型）以N标记
  },
  "TableName": "app-entity"      <---添加项目到app-entity表
};
db.putItem(params, function(err) {      <---调用DynamoDB的putItem 操作
  if (err) {      <---处理错误
    console.error('error', err);
  } else {
    console.log('success');
  }
});
```

第一步是添加数据到nodetodo。

## 10.5.1 添加一个用户

可以调用nodetodo user-add <uid> <email> <phone> 命令来添加用户到nodetodo。在Node.js中，可以用代码清单10-3中的代码实现这一点。

代码清单10-3 nodetodo：添加一个用户（index.js）

```
if (input['user-add'] === true) {
  var params = {
    "Item": {
      "uid": {"S": input['<uid>']},      <---项目包含所有属性。键也是属性，所以
      <---必须在添加数值时告诉DynamoDB 哪些属性是键      <---uid 属性为字符串类型，包含ui
      d 的参数值
    }
  }
}
```

```

    "email": {"S": input['<email>']},      <--email 属性类型为字符串, 包含电
电子邮件的参数值
    "phone": {"S": input['<phone>']},      <--phone 电话属性为字符串类型, 包
含电话号码的参数值
  },
  "TableName": "todo-user",              <--指定用户表
  "ConditionExpression": "attribute_not_exists(uid)"    <--如果对同样的键
执行两次putItem 操作, 数据会被替换。配合ConditionExpression 条件表达式可用仅在
不存在同样键的情况下putItem 写入项目
};
db.putItem(params, function(err) {      <--调用DynamoDB的putItem 操作
  if (err) {
    console.error('error', err);
  } else {
    console.log('user added with uid ' + input['<uid>']);
  }
});
}

```

调用AWS API时, 需要完成下面的操作。

(1) 创建一个JavaScript对象(映射), 使用需要的参数(**params** 变量)作为对象的属性。

(2) 调用AWS SDK的函数。

(3) 检查响应消息中是否包含错误, 或者处理返回的数据。

这样, 如果需要添加一个任务而不是一个用户的话, 只需要改变**params** 的内容即可。

## 10.5.2 添加一个任务

可以调用**nodetodo task-add <uid><description> [<category>][--duedate=<yyyymmdd>]** 来添加任务到**nodetodo**。在Node.js中, 使用代码清单10-4中的代码实现这一功能。

代码清单10-4 nodetodo: 添加一个任务 (index.js)

```

if (input['task-add'] === true) {
  var tid = Date.now();      <--基于当前时间戳来创建任务ID (tid)
  var params = {
    "Item": {
      "uid": {"S": input['<uid>']},
      "tid": {"N": tid.toString()},    <--tid 属性为数值类型, 包含tid 的值
      "description": {"S": input['<description>']},
      "created": {"N": moment().format("YYYYMMDD")}    <--创建的属性为数值
类型 (格式如20150525)
    },
    "TableName": "todo-task",    <--指定任务表
    "ConditionExpression":
      "attribute_not_exists(uid) and attribute_not_exists(tid)"
  };
  if (input['--dueat'] !== null) {    <--如果设置了可选的dueat参数, 添加该值
到项目中
    params.Item.due = {"N": input['--dueat']};
  }
  if (input['<category>'] !== null) {    <--如果设置了可选的category参数,
添加该值到项目中
    params.Item.category = {"S": input['<category>']};
  }
  db.putItem(params, function(err) {    <--调用DynamoDB 的putItem 操作
    if (err) {
      console.error('error', err);
    } else {
      console.log('task added with tid ' + tid);
    }
  });
}
}

```

现在用户就可以添加用户和任务到nodetodo。如果还能读取出所有这些数据会不会更棒？



## 10.6 获取数据

DynamoDB是一个键值存储。键通常是从这类存储中获得数据的唯一的入口。在设计DynamoDB的数据模型的时候，用户必须在创建表的时候注意这个限制（10.4节中介绍过如何创建表）。如果只能使用一个键来查询数据，不久你就会遇到麻烦。幸运的是，DynamoDB提供了两种其他的方法来查询项目：一个二级索引查询和扫描操作。你将从简单的主键查询开始，然后继续了解更复杂的数据获取方式。

### DynamoDB Streams

在修改数据之后，DynamoDB允许你马上查询到修改后的数据。DynamoDB Streams捕获对表项目的所有写操作（创建、修改和删除），并且按照顺序记录对给定分区键的项目的修改操作：

- 和应用需要轮询数据库来捕捉数据变更相比，DynamoDB Streams以更优雅的方法满足同样需求。
- DynamoDB Streams可以帮助把表里的数据变更复制到缓存里。
- DynamoDB Streams可以帮助把数据跨区域复制到另外一张表里。

### 10.6.1 提供键来获取数据

最简单的获取数据的方式就是使用它的主键查找数据。`getItem` SDK操作可以从DynamoDB获取一个单独的项目：

```
var params = {
  "Key": {
    "attr1": {"S": "val1"}    <---指定组成键的属性
  },
  "TableName": "app-entity"
};
db.getItem(params, function(err, data) {    <---调用DynamoDB 表的getItem 操作
  if (err) {
    console.error('error', err);
  } else {
```

```

    if (data.Item) {      <---检查是否找到项目
        console.log('item', data.Item);
    } else {
        console.error('no item found');
    }
}
});

```

**nodetodo user <uid>** 命令必须使用用户ID (**uid**) 来获取一个用户的信息。具体实现的Node.js AWS SDK的代码如代码清单10-5所示。

代码清单10-5 nodetodo: 提取一个用户 (index.js)

```

function mapUserItem(item) {      <---Helper 帮助函数转化DynamoDB返回的结果
    return {
        "uid": item.uid.S,
        "email": item.email.S,
        "phone": item.phone.S
    };
}

if (input['user'] === true) {
    var params = {
        "Key": {
            "uid": {"S": input['<uid>']}      <---按照主键uid 查找用户
        },
        "TableName": "todo-user"      <---指定用户表
    };
    db.getItem(params, function(err, data) {      <---调用DynamoDB 的getItem 操作
        if (err) {
            console.error('error', err);
        } else {
            if (data.Item) {      <---检查是否找到满足主键参数值的数据
                console.log('user', mapUserItem(data.Item));
            } else {
                console.error('user not found');
            }
        }
    });
}
}

```

用户还可以使用`getItem`操作使用分区键和排序键的组合主键来查询数据。唯一的不同是，组合`Key`有两个条目而不是之前的一个，`getItem`返回一个项目或者没有项目返回。如果希望一次查询获取多个项目，就必须使用DynamoDB的查询API。

## 10.6.2 使用键和过滤来查询

如果希望返回一组项目而不只是一个项目，就只能使用DynamoDB的查询API。只有在表里有一个分区键和排序键的时候，才能使用分区键返回多个项目。否则，分区键仅返回一个项目。使用`query`查询SDK从DynamoDB返回一组项目的操作如下：

```
var params = {
  "KeyConditionExpression": "attr1 = :attr1val AND attr2 = :attr2val",
  ←--键必须满足的条件。如果有分区键和排序键条件，就必须使用AND 操作符。分区键允许的操作符为=。排序键允许使用的操作符为=、>、<、>=、<=、BETWEEN... AND...
  "ExpressionAttributeValues": {
    ":attr1val": {"S": "val1"},      ←--在表达式中引用动态值
    ":attr2val": {"N": "2"}        ←--始终指定正确的类型（S、N 和B）
  },
  "TableName": "todo-task"
};
db.query(params, function(err, data) {      ←--调用DynamoDB的query 操作
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
  }
});
```

`query`操作还允许指定一个可选的`FilterExpression`过滤表达式。`FilterExpression`过滤表达式的语法类似`KeyConditionExpression`键条件表达式，但是过滤没有使用索引。过滤条件作用在所有满足`KeyConditionExpression`查询条件的返回结果上。

要返回一个特定用户的所有任务，就必须查询DynamoDB。一个任

务的主键是uid 的分区键部分和tid 的排序键部分的组合。要返回所有用户的任务，KeyConditionExpression 只需要主键的分区键部分做相等运算。Nodetodo task-ls <uid> [<category>] [--overdue|--due|--withoutdue|--futuredue] 的代码实现如代码清单10-6所示。

代码清单10-6 nodetodo: 提取任务 (index.js)

```
function getValue(attribute, type) {      <---Helper 函数访问
可选的属性
    if (attribute === undefined) {
        return null;
    }
    return attribute[type];
}

function mapTaskItem(item) {      <---Helper 函数对DynamoDB的返回结果进行转化
    return {
        "tid": item.tid.N,
        "description": item.description.S,
        "created": item.created.N,
        "due": getValue(item.due, 'N'),
        "category": getValue(item.category, 'S'),
        "completed": getValue(item.completed, 'N')
    };
}

if (input['task-ls'] === true) {
    var now = moment().format("YYYYMMDD");
    var params = {
        "KeyConditionExpression": "uid = :uid",      <---主键查询。任务表使用分区键
和排序键。查询中只使用了分区键，将返回所有满足条件的分区键
        "ExpressionAttributeValues": {
            ":uid": {"S": input['<uid>']}      <---查询属性必须以这样的格式传入
        },
        "TableName": "todo-task"
    };
    if (input['--overdue'] === true) {
        params.FilterExpression = "due < :yyyymmdd";      <---过滤器不使用索引。在
所有满足主键查询的返回结果里应用过滤器
        params.ExpressionAttributeValues[':yyyymmdd'] = {"N": now};      <---可用
使用逻辑操作符组合多个过滤器
    } else if (input['--due'] === true) {
        params.FilterExpression = "due = :yyyymmdd";
        params.ExpressionAttributeValues[':yyyymmdd'] = {"N": now};
    } else if (input['--withoutdue'] === true) {
```

```

    params.FilterExpression = "attribute_not_exists(due)";    <--在缺少相
应的属性的时候（与attribute_exists 相反） 返回attribute_not_exists(due)为true
  } else if (input['--futuredue'] === true) {
    params.FilterExpression = "due > :yyyyymmdd";
    params.ExpressionAttributeValues[':yyyyymmdd'] = {"N": now};
  }
  if (input['<category>'] !== null) {
    if (params.FilterExpression === undefined) {
      params.FilterExpression = '';
    } else {
      params.FilterExpression += ' AND ';    <--多个过滤器可以和逻辑运算符组
合
    }
    params.FilterExpression += 'category = :category';
    params.ExpressionAttributeValues[':category'] = {"S": input['<category
>']});
  }
  db.query(params, function(err, data) {    <--调用DynamoDB的查询操作
    if (err) {
      console.error('error', err);
    } else {
      console.log('tasks', data.Items.map(mapTaskItem));
    }
  });
}

```

查询API调用会产生以下两个问题。

- 如果满足主键查询的结果集很大，过滤性能可能很慢。过滤条件没有使用索引：每个返回的项目都必须进行检查。想象一下，如果DynamoDB里面保存的是股票价格，使用了分区键和排序键：分区键值是AAPL（Apple苹果公司股票代码），排序键是时间戳，你可以查询到苹果公司（AAPL）在2010年1月1日到2015年1月1日之间的所有股价信息。但是如果只希望返回每周一的价格，你需要定义过滤条件，仅返回结果集中的20%的数据。那将浪费很多资源！
- 可以只查询主键。不可能只返回所有用户的属于某个特定的目录的任务，因为不能针对category 属性进行查询。

我们可以使用二级索引解决这些问题。下面我们就来看一下二级索引是如何工作的。

## 10.6.3 更灵活地使用二级索引查询数据

二级索引是DynamoDB自动维护的原来那张表的映射。可以像查询一张表的主键那样查询一个二级索引。可以把全局二级索引想象为一个DynamoDB表的只读副本，DynamoDB服务会自动更新索引表里的数据：一旦你修改了主表里的数据，所有的索引就会异步地更新（最终一致！）。图10-2展示了二级索引的工作方式。

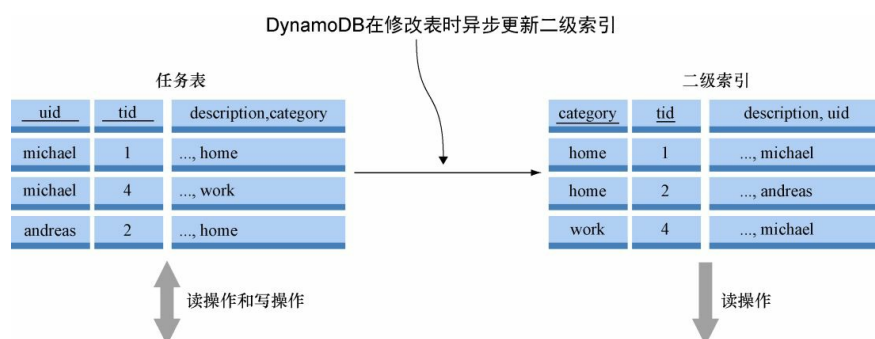


图10-2 二级索引包含了用户表数据的一份副本，以便基于另外的键来快速查询

使用二级索引需要付出成本：索引需要存储容量（和源表的存储成本一样）。你必须为索引配置额外的写容量单位，因为每个对主表的写入操作都会造成到二级索引的写操作。

DynamoDB的一个很大的优势是可以根据工作负载动态配置性能的容量单位。如果你的一个索引接到了大量的读请求，可以增加索引的读容量单位。还可以精细化地调整表和索引的性能。10.9节将介绍如何操作。

回到nodetodo的例子。为了实现按照目录来读取任务列表的需求，你将为todo-task表添加一个二级索引。这将允许你按照目录来查询。一样用到了分区键和排序键：分区键是category属性，排序键是tid属性。索引还需要一个名字：category-index。你可以查看在nodetodo的代码目录下的README.md找到下面的命令：

```
$ aws dynamodb update-table --table-name todo-task \      <---在创建表的时候
添加全局二级索引
--attribute-definitions AttributeName=uid,AttributeType=S \
AttributeName=tid,AttributeType=N \
```

```

AttributeName=category,AttributeType=S \      <--添加一个目录category 属性,
因为索引将用到该属性
--global-secondary-index-updates '[{\
"Create": {\      <--创建新的二级索引
"IndexName": "category-index", \
"KeySchema": [{ "AttributeName": "category", "KeyType": "HASH"}], \      <--c
ategory 属性为主键的分区部分, tid属性是排序部分
{ "AttributeName": "tid", "KeyType": "RANGE"}], \
"Projection": { "ProjectionType": "ALL"}, \      <--把所有的属性映射到索引
"ProvisionedThroughput": { "ReadCapacityUnits": 5, \
"WriteCapacityUnits": 5}\
}]']

```

创建全局二级索引的需要一定的时间, 可以使用命令查看索引是否进入active活动状态:

```

$ aws dynamodb describe-table --table-name=todo-task \
--query "Table.GlobalSecondaryIndexes"

```

代码清单10-7展示了使用query 操作实现nodetodo task-la <category> [--overdue|...] 的代码。

代码清单10-7 nodetodo: 从目录索引获取任务 (index.js)

```

if (input['task-la'] === true) {
  var now = moment().format("YYYYMMDD");
  var params = {
    "KeyConditionExpression": "category = :category",      <--查询索引一样需
    要以主键作为查询条件
    "ExpressionAttributeValues": {
      ":category": {"S": input['<category>']}
    },
    "TableName": "todo-task",
    "IndexName": "category-index"      <--但是你可以指定使用哪个索引查询
  };
  if (input['--overdue'] === true) {
    params.FilterExpression = "due < :yyyyymmdd";      <--过滤器的使用方法和基
    于主键查询一样
    params.ExpressionAttributeValues[':yyyyymmdd'] = {"N": now};
  }
  [...]
}

```

```

db.query(params, function(err, data) {
  if (err) {
    console.error('error', err);
  } else {
    console.log('tasks', data.Items.map(mapTaskItem));
  }
});
}

```

但是查询也会有无法满足的情况：无法提取所有的用户。下面来了解一下表扫描可以帮助我们实现哪些功能。

## 10.6.4 扫描和过滤表数据

有些查询只针对主键是无法满足的；相反，你需要遍历表里的所有项目。这种操作效率很低，但是在某些情况下是可以接受的。

DynamoDB提供scan 操作遍历表里的所有项目。

```

var params = {
  "TableName": "app-entity",
  "Limit": 50      <--指定每次扫描操作返回的项目的最大数量
};
db.scan(params, function(err, data) {      <--调用DynamoDB的scan 扫描操作
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
    if (data.LastEvaluatedKey !== undefined) {      <--检查是否还有更多项目可供扫描
      console.log('more items available');
    }
  }
});

```

代码清单10-8展示了`nodetodo user-ls [--limit=<limit>] [--next=<id>]`的实现。它使用了分页的机制来防止返回太多的项目。



```

if (input['user-ls'] === true) {
  var params = {
    "TableName": "todo-user",
    "Limit": input['--limit']      <-- 每次扫描操作返回的项目数量
  };
  if (input['--next'] !== null) {
    params.ExclusiveStartKey = {      <-- 命名的参数包含上一次获取的键值
      "uid": {"S": input['--next']}
    };
  }
  db.scan(params, function(err, data) {      <-- 调用DynamoDB 的scan 操作
    if (err) {
      console.error('error', err);
    } else {
      console.log('users', data.Items.map(mapUserItem));
      if (data.LastEvaluatedKey !== undefined) {      <-- 检查是否已经扫描返回
        了所有的项目
        console.log('more users available with
          ➡ --next=' + data.LastEvaluatedKey.uid.S);
      }
    }
  });
}

```

**scan** 操作读取表里的所有项目。在本例中没有过滤任何数据，但是可以配合使用**FilterExpression** 过滤表达式。注意不要频繁使用**scan** 操作——它很灵活但不够高效。

## 10.6.5 最终一致地数据提取

DynamoDB不支持传统数据库支持的事务。不能在一个事务中修改（即创建、更新和删除）多个项目——项目是DynamoDB中的原子操作单元。

另外，DynamoDB是最终一致的。这意味着如果创建了一个项目（版本1），更新该项目到第二个版本，然后马上读取该项目，你可能还会看到旧的版本1的数据；如果等待一下然后获取该项目，会看到版

本2。图10-3展示了这个过程。不同的服务器服务用户的查询请求，但是请求到达的DynamoDB服务器上可能没有最新版本的数据。

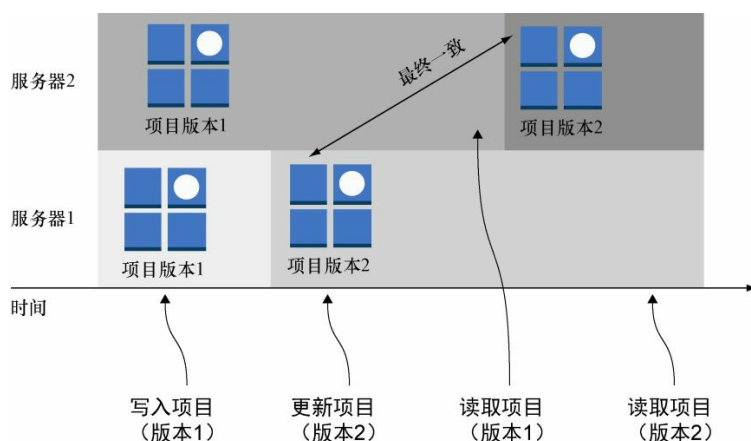


图10-3 在写操作更新到所有后端服务器之前，最终一致的读操作可能返回旧的数据

可以设置"**ConsistentRead**":true 的选项来向DynamoDB请求强一致的读操作，以避免读取到最终一致的数据。**getItem**、**query** 和 **scan** 等操作支持强一致的读选项。但是强一致的读操作要花更多的时间，和最终一致的读操作相比会消耗更多的读单位容量。从全局二级索引读到的数据永远是最终一致的，因为索引自身是最终一致的。

## 10.7 删除数据

和`getItem`操作类似，`deleteItem`操作需要指定要删除的项目的主键值。根据表使用的主键类型的不同（分区主键还是分区和排序组合主键），你需要指定一个或者两个属性。

调用`nodetodo user-rm <uid>`可以删除一个用户。在Node.js中，如代码清单10-9所示。

代码清单10-9 nodetodo: 移除用户（index.js）

```
if (input['user-rm'] === true) {
  var params = {
    "Key": {
      "uid": {"S": input['<uid>']}    <--按照分区键识别一个项目
    },
    "TableName": "todo-user"        <--指定user 用户表
  };
  db.deleteItem(params, function(err) {    <--调用DynamoDB 的deleteItem 属性
    if (err) {
      console.error('error', err);
    } else {
      console.log('user removed with uid ' + input['<uid>']);
    }
  });
}
```

删除一个任务的操作也类似：`nodetodo task-rm <uid> <tid>`。唯一的不同是需要使用表的名称、提供分区键和排序键来识别一个特定的项目，如代码清单10-10所示。

代码清单10-10 nodetodo: 移除一个任务（index.js）

```
if (input['task-rm'] === true) {
  var params = {
    "Key": {
      "uid": {"S": input['<uid>']},
```

```
    "tid": {"N": input['<tid>']}    <--使用分区键和排序键识别一个项目
  },
  "TableName": "todo-task"    <--指定任务表
};
db.deleteItem(params, function(err) {
  if (err) {
    console.error('error', err);
  } else {
    console.log('task removed with tid ' + input['<tid>']);
  }
});
}
```

现在我们学会了创建、读取和删除DynamoDB中的项目，唯一没讲的操作就是更新数据。

## 10.8 修改数据

更新项目可以使用`updateItem`操作。用户必须通过其键来识别你想更新的项目，用户也可以提供一个`UpdateExpression`更改表达式来指定想要完成的更新操作。可以使用更新操作的一种或者几种组合。

- 使用`SET`来覆盖或者创建一个新的属性，如`SET attr1=:attr1val`, `SET attr1=attr2+:attr2val`, `SET attr1 = :attr1val`, `attr2 = :attr2val`。
- 使用`REMOVE`来删除一个属性，如`REMOVE attr1`, `REMOVE attr1, attr2`。

在`nodetodo`应用中，可以使用`nodetodo task-done <uid> <tid>`来标记一项任务为完成。为了实现这个功能，需要修改任务的项目，如代码清单10-11中的Node.js代码所示。

代码清单10-11 nodetodo: 修改任务为完成 (index.js)

```
if (input['task-done'] === true) {
  var now = moment().format("YYYYMMDD");
  var params = {
    "Key": {
      "uid": { "S": input['<uid>']},      <--使用分区键和排序键识别一个项目
      "tid": { "N": input['<tid>']}
    },
    "UpdateExpression": "SET completed = :yyymmdd",      <--定义要修改哪个属性
    "ExpressionAttributeValues": {
      ":yyymmdd": {"N": now}      <--必须以这样的格式定义属性修改
    },
    "TableName": "todo-task"
  };
  db.updateItem(params, function(err) {      <--调用DynamoDB 的updateItem修改属性操作
    if (err) {
      console.error('error', err);
    } else {
      console.log('task completed with tid ' + input['<tid>']);
    }
  });
}
```

}

## 10.9 扩展容量

在创建一个DynamoDB的表或者索引的时候，必须配置吞吐量。吞吐量分为读容量单位和写容量单位。DynamoDB使用ReadCapacityUnits 和WriteCapacityUnits 来分别指定表或者全局二级索引的吞吐量性能。但是，容量单位（Capacity Unit）如何定义呢？我们通过命令行接口来体验一下：

```
$ aws dynamodb get-item --table-name todo-user \
--key '{"uid": {"S": "michael"}}' \
--return-consumed-capacity TOTAL \      <---告诉DynamoDB 返回使用的容量单位
--query "ConsumedCapacity"
{
  "CapacityUnits": 0.5,      <---getItem 操作使用了0.5 个容量单位
  "TableName": "todo-user"
}
$ aws dynamodb get-item --table-name todo-user \
--key '{"uid": {"S": "michael"}}' \
--consistent-read --return-consumed-capacity TOTAL \      <---强一致的读操作
--query "ConsumedCapacity"
{
  "CapacityUnits": 1.0,      <---需要两倍的容量单位
  "TableName": "todo-user"
}
```

关于吞吐量的消耗的更多信息如下所示。

- 和强一致的读操作相比，最终一致的读操作消耗一半的读容量单位
- 如果一个项目的大小不超过4 KB，强一致的getItem 操作消耗一个读容量单位。如果项目超过4 KB大小，你需要额外的读容量单位。你可以使用roundUP (itemSize/4) 计算需要多少单位。
- 对4 KB大小的query 进行一个强一致的查询，需要消耗一个读容量单位。这意味着如果查询返回10个项目，每个项目的大小是2 KB，总的项目大小为20 KB，会需要5个读容量单位。和10个getItem 操作相比很不一样，getItem 的操作将使用10个读容量单位。
- 对1 KB大小的项目的1个写操作，会消耗一个写容量单位。如果项目超过1 KB，可以使用roundUP (itemSize) 取整计算所需要的

写容量单位。

如果不熟悉容量单位的概念，可以使用 AWS 简单月度成本计算器，提供读写负载的细节，它会帮用户计算出所需的容量单位。

表和索引的吞吐量配置以每秒为单位来计算。如果使用 `ReadCapacityUnits=5` 来设置每秒5个读容量单位，并且该表的项目的大小不超过4 KB，每秒钟就可以对该表进行5个强一致的 `getItem` 请求。如果用户提交超过配置容量的请求，DynamoDB会拒绝超出的请求。

监控读容量和写容量单位的使用量很重要。幸运的是，DynamoDB 服务每分钟会向CloudWatch服务发送有用的指标。要查看这些指标，打开AWS管理控制台，访问到DynamoDB服务的页面，选择其中的 `todo-user` 表，然后选择“指标”选项卡。图10-4展示了 `todo-user` 表的 CloudWatch 指标。

可以随时修改表的预配置吞吐量，但是每天只能降低4次表的吞吐量单位。

#### 资源清理

在本节结束时别忘了删除DynamoDB的表。使用AWS管理控制台进行删除操作。



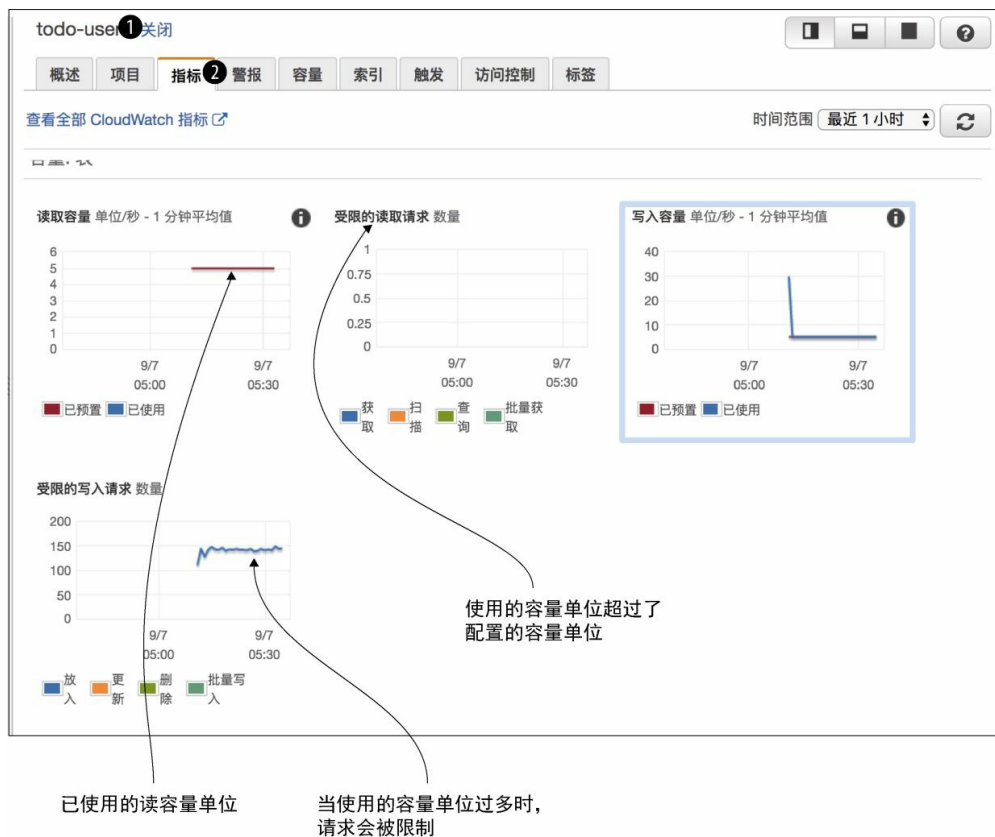


图10-4 监控配置和使用的DynamoDB表的容量单位

## 10.10 小结

- DynamoDB是一个NoSQL的数据库服务，用户不再有任何运维管理的负担，可以很方便地扩展，在多种情况下用作应用程序的后端存储。
- 对DynamoDB表的数据的查询基于键。只有在知道键值的情况下才能对一个分区键进行查询。但是DynamoDB还支持分区键和排序键，这在分区键之外提供了额外的排序键。
- 可以提供键给`getItem`操作以返回一个项目。
- 如有需要，可以指定强一致的读操作（`getItem`、`query`和`scan`）。对全局二级索引提交的读操作永远是最终一致的。
- DynamoDB不提供SQL接口。不同的是，需要使用SDK来让应用程序和DynamoDB通信。这也意味着必须修改现有的应用程序的代码，才能让它使用DynamoDB。
- DynamoDB使用表达式来完成更加复杂的数据库交互，例如，在需要修改一个项目的时候。
- 如果想要为表和索引配置足够的容量单位，监控使用的读容量和写容量单位就非常重要。
- DynamoDB按照使用的每GB存储容量和预配置的读容量和写容量单位来收费。
- 可以使用`query`操作来查询主键或者二级索引。
- `scan`操作很灵活，但是效率不高，尽量避免使用该操作。

## 第四部分 在AWS上搭架构

Amazon.com的CTO沃纳·威格尔（Werner Vogels）经常被引用的一句名言是：“每个事物在任何时间都可能失效。”这句话是AWS背后的一个重要的理念。不要试图让自己的系统牢不可，这是一个无法达到的目标，而AWS就是为了应对失效而设计的。硬盘驱动器会出现故障，因此S3服务将数据存储多个硬盘上，以防止数据丢失。计算机硬件会发生故障，如果需要虚拟服务器能够被自动在另一台服务器上重新启动。数据中心也可能失效，因此每个区域有多个数据中心，这些数据中心可以同时按照需要而使用。

在本书的这一部分，读者将学习如何通过使用正确的工具和架构，防止你的运行在AWS上的应用程序运行中断。下面的表中列出了最重要的服务和故障处理的方法。

	描述	示例
容错	服务可以从失败自动恢复，无须停机	S3（对象存储）、DynamoDB（NoSQL数据库）、Route 53（DNS）
高可用	服务可以自动从某些故障中恢复，只需要一个短暂的停机时间	RDS（关系数据库）、EBS（网络存储）
手动的失效处理	服务不能够实现默认的故障恢复，但是提供了用于建立高可用基础设施的工具	EC2（虚拟服务器）

面向失效设计是AWS的一个基本原则，另一个充分利用云计算的弹性。你还将学习在AWS之上如何基于当前的工作和架构的可靠性要求增加虚拟服务器的数量。第11章提供了关于关于单一服务器和数据中心失效风险基础知识。第12章讨论如何解耦系统以提高系统的可靠性：

采用同步解耦以及利用负载均衡器的帮助，或者过亚马逊AWS的分布式队列服务SQS实现异步解耦，来建立一个容错系统。第13章涵盖了基于EC2实例（非默认的容错），设计一个容错的Web应用程序。第14章是关于弹性和自动扩展的，将学习根据计划或当前系统负载来扩展容量。

# 第11章 实现高可用性：可用区、自动扩展以及CloudWatch

## 本章主要内容

- 使用CloudWatch实现虚拟服务器失效告警
- 理解AWS区域下的可用区
- 使用自动扩展以保证虚拟服务器的正常运行
- 分析灾难恢复的需求

在本章中，我们将介绍如何基于EC2实例搭建一个高可用性架构。需要强调的一点是，在默认情况下虚拟服务器没有为高可用而做设置。下面罗列的场景将会导致虚拟服务器的停机。

- 虚拟服务器因为软件问题（虚拟服务器的操作系统）失败。
- 发生在宿主机服务器上的软件故障，导致虚拟服务器的崩溃（宿主机服务器的操作系统或者虚拟化层的问题）。
- 物理设备上的计算、存储以及网络等硬件的故障。
- 数据中心中的虚拟服务器所依赖的资源的故障，如网络连接、供电以及制冷系统等。

例如，如果物理主机上的计算机硬件出现故障，所有的运行在该主机上的EC2实例将会失效。如果用户在一台受故障影响的虚拟服务器上运行自己的应用程序，这个应用将无法正常运行。直到有人或许用户自己在另一台物理主机上启动一个新的虚拟服务器。为了避免这些情况的出现，你应当瞄准具有高可用性的虚拟服务器，可以在无须人工干预的情况下自动从故障中恢复。

### 示例都包含在免费套餐中

本章中的所有示例都包含在免费套餐中。只要你不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于你为学习本书刚刚创建的全新AWS账户，并且在你的AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

高可用性通常被描述为系统的运行几乎没有停机时间。即使发生故障，系统也能够以较大的可能性继续提供服务。虽然需要短暂的中断以便系统从故障中恢复，但是这个过程不需要人工交互。哈佛研究团队（Harvard Research Group, HRG）使用AEC-2分类法对高可用性做出了定义，在一年以内需要满足99.99%的正常运行。

#### 高可用性与容错

一个高可用性系统可以在较短的停机时间内自动从故障中恢复。相比之下，容错系统要求系统提供的服务不会因为一个组件失效而无法提供服务。第13章将展示如何构建一个容错系统。

AWS提供了构建基于EC2实例的高可用系统的工具。

- 利用CloudWatch监控虚拟服务器的运行状况。如果需要，自动触发故障恢复。
- 通过使用多个隔离的数据中心（在AWS称作一个区域内的可用区，简称AZ）搭建高可用的基础架构。
- 使用自动扩展（auto-scaling）确保拥有一定数量的虚拟服务器用以自动替换失效的实例。

## 11.1 使用CloudWatch恢复失效的服务器

AWS的EC2服务会自动地检查每个虚拟服务器的状态。这种检查每一分钟都将进行，活跃状态是CloudWatch的监测指标。AWS CloudWatch是一个提供监测指标、日志和告警的服务。在第9章中我们已经了解使用CloudWatch以获得关系数据库实例当前的负荷。图11-1展示了如何在EC2实例的详细信息页面手动设置CloudWatch告警，告警的信息来自EC2实例的系统检查。



图11-1 基于系统检查的指标建立CloudWatch告警，一旦EC2的实例失效就会触发自动恢复

系统状态检查通常是检测网络连接、供电或者物理主机上的软件或硬件是否存在问题。AWS需要通过系统检查去发现失效并启动修复。一个常常用于解决此类故障的策略是将出现故障的虚拟服务器恢复到另一台物理主机之上。

图11-2展示了停机处理的流程对虚拟服务器的影响。

- (1) 物理服务器的失效导致运行其上的虚拟服务器失效。
- (2) EC2服务检测到运行中断并报告给CloudWatch的度量。

- (3) CloudWatch告警触发了对于虚拟服务器的恢复。
- (4) 在另外一台物理服务器上启动这个虚拟服务器。
- (5) EBS卷和弹性IP将被连接到新的虚拟服务器并保持原样。

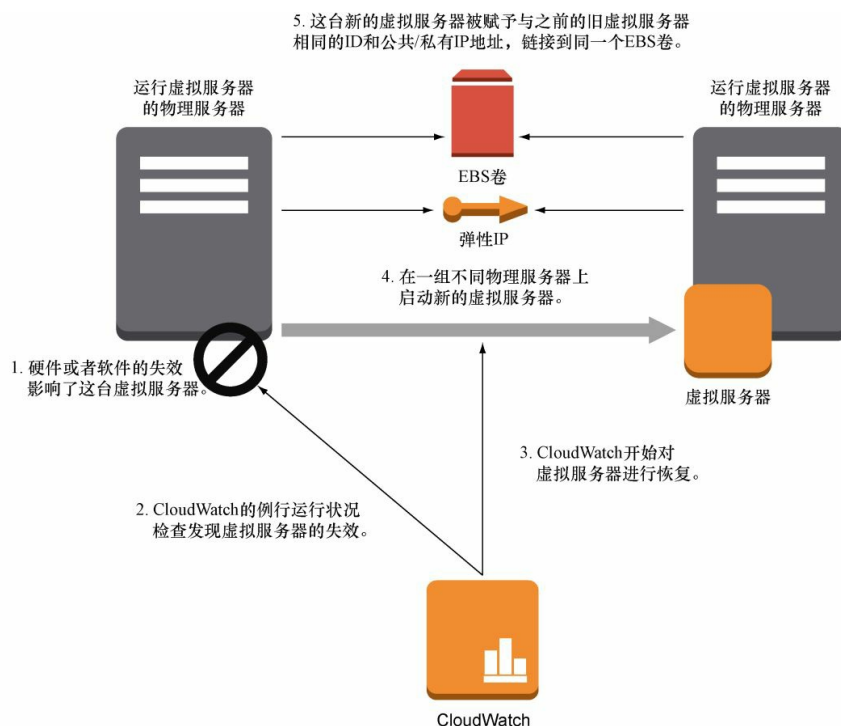


图11-2 在硬件失效的情况下，CloudWatch触发对虚拟服务器的恢复

完成恢复以后，新的被运行的虚拟服务器拥有相同的ID和私有IP地址。在EBS卷上的数据，这是一种网络存储，也同样被恢复。因为拥有的是同一个EBS卷，数据不会被丢失。虚拟服务器自身的本地磁盘（实例存储）无法通过CloudWatch告警触发的处理流程而恢复。如果原有的虚拟服务器设置有弹性IP，新的服务器也会拥有同样的公有IP地址。

#### 恢复EC2实例的要求

如果你打算使用这个恢复的功能，虚拟服务器必须满足以下的条件。

- 它必须运行在一个虚拟专网（VPC）之内。
- 实例家族必须是c3（计算优化）、c4（计算优化）、m3（通用）、r3（内存优化）或者t2（突发性能优化）。早期的实例家族不被支持（如t1）。
- EC2实例只能使用EBS卷，因为这样可以保证在实例恢复以后数据不会丢失。



## 11.1.1 建立一个CloudWatch告警

一个CloudWatch告警由以下部分组成。

- 一组监控数据的指标（运行状况检查、CPU利用率等）。
- 一个规则定义在一段时间基于统计函数的阈值。
- 当告警状态改变时触发的动作（例如，如果告警状态改变触发一个EC2实例恢复）。

下列的状态可以被用来告警。

- **OK** ——一切状态都很好，还没有达到阈值。
- **INSUFFICIENT\_DATA** ——没有足够的数据来评估告警。
- **ALARM** ——有东西发生了故障，告警的门槛已经被越过。

当你需要监视虚拟服务器的运行状况，并需要当宿主的系统出现故障进行恢复的时候，可以使用像代码清单11-1所示的一个CloudWatch告警进行设置。这个代码清单是从CloudFormation模板中摘录出来的。

代码清单11-1创建了一个CloudWatch告警，设置的基础来自一个名为**StatusCheckFailed\_System**（通过**MetricName** 属性链接）的度量标准。该指标包括了由EC2服务每分钟所进行的虚拟服务器系统状态检查的结果。如果检查失败，一个具有数值1的测量点被加入到**StatusCheckFailed\_System** 指标中。因为EC2服务会发布该指标，因此**Namespace** 被称为**AWS/EC2** 以及**Dimension** 的维度使用了虚拟服务器的ID。

CloudWatch告警指标检查的**Period** 属性设定为每60 s一次。如在**EvaluationPeriods** 中定义的那样，告警服务将会检查最后的5个周期的状态，在这种情况下这意味着最后的5 min。检查将在这期间运行**Statistic** 中指定的统计函数。在这种情况下对于统计函数的结果，最小值函数将会用**ComparisonOperator** 将其与**Threshdd** 进行比较。如果结果是否定的，在**AlarmActions** 中定义的告警动作将会被执行——在代码清单11-1中，虚拟服务器的恢复是EC2实例的内置动作。

```
[...]
"RecoveryAlarm": {      <--创建一个CloudWatch 告警用以监控虚拟服务器的运行状况

  "Type": "AWS::CloudWatch::Alarm",
  "Properties": {
    "AlarmDescription": "Recover server when underlying hardware fails.",
    "Namespace": "AWS/EC2",      <--运行状况检查的指标名称，EC2实例包含的系统故障检查的名称
    "MetricName": "StatusCheckFailed_System",      <--监控器指标由EC2 服务提供，命名空间用的是AWS/EC2
    "Statistic": "Minimum",      <--统计函数被用于检测的指标，即使最小的状态检查失效也会得到通知
    "Period": "60",      <--统计函数用来计算应用的时间，以秒为单位，并且必须是60的倍数
    "EvaluationPeriods": "5",      <--用于将数据与阈值进行比较的周期数
    "ComparisonOperator": "GreaterThanThreshold",      <--将统计函数的输出结果与阈值进行比较的操作
    "Threshold": "0",      <--阈值触发告警
    "AlarmActions": [{      <--告警情况下采取的动作。使用针对EC2 实例的预定义的恢复操作
      "Fn::Join": [ "", [ "arn:aws:automate:", { "Ref": "AWS::Region" } ], ":ec2:recover" ] ]
    },
    "Dimensions": [{ "Name": "InstanceId", "Value": { "Ref": "Server" } }]
  }
}
[...]
```

总之，虚拟服务器的状态由AWS每分钟检查一次。告警服务检查 `StatusCheckFailed-System` 指标。如果有连续的5个失效的报告，告警将被触发。

## 11.1.2 基于CloudWatch对虚拟服务器监控与恢复

假定你的团队正在你的开发过程中采用敏捷流程。为了加速这个流程，你的团队决定采用自动化的软件测试、构建和部署。于是你被要求

设立一个持续集成服务器（CI服务）。你为此选择了使用Jenkins，这是一个用Java编写的运行在servlet容器（如Apache Tomcat）上的开源应用。因为你所使用的环境具有“基础设施即是代码”的特性，你打算在你的基础设施之上做出调整以及部署Jenkins。

Jenkins服务器是一个典型的设置高可用性的使用场景。这是你的基础设施当中非常重要的一个部分。一旦这个服务出现停机故障，你的同事将无法测试和部署新的软件。因为系统故障而导致的系统恢复只会产生很短的停机时间，而且故障恢复不会破坏你的业务的情况下，其实你并不需要一个容错的系统。

在这个例子里面，你将按照下面的步骤操作。

- （1）在云计算的环境里面建立一个虚拟网络（VPC）。
- （2）在VPC中启动一个虚拟服务器，并通过初始化启动程序（bootstrap）自动安装Jenkins。
- （3）创建一个CloudWatch告警服务，监控虚拟服务器的运行状况。

我们将帮助你通过CloudFormation模板的帮助完成这些步骤。你可以通过GitHub以及S3找到用于这个例子的CloudFormation模板。

我们在第11章中谈到的recovery.json可以在S3找到，文件位于[https://s3.amazonaws.com/ awsinaction/chapter11/recovery.json](https://s3.amazonaws.com/awsinaction/chapter11/recovery.json)。

关于学习和了解更多关于Jenkins的内容，可以参考其官方文档。

下面的命令启动一个含有EC2实例与CloudWatch告警触发服务器失效恢复的CloudFormation模板。使用由8~40个字符和数字组成的密码替换\$Password。一个Jenkins服务器将会自动安装在一个虚拟服务器上：

```
$ aws cloudformation create-stack --stack-name jenkins-recovery \  
--template-url https://s3.amazonaws.com/  
awsinaction/chapter11/recovery.json \  
--parameters ParameterKey=JenkinsAdminPassword,ParameterValue=$Password
```

CloudFormation模板包含了私有网络和安全的配置。但是该模板最重要的部分是下面的这些。

- 虚拟服务器的用户数据包含一个bash的脚本用以在启动期间安装Jenkins。
- 公有IP地址被分配给新的虚拟服务器，你可以在服务器被恢复以后使与之前相同的IP地址去访问它。
- CloudWatch告警服务基于EC2服务发布的系统状态的指标。

代码清单11-2展示了CloudFormation模板中重要的部分。

代码清单11-2 在EC2实例上启动运行具有告警恢复能力的Jenkins CI服务器

```
[...]
"ElasticIP": {      <--使用弹性IP 提供的公有IP 地址将会在服务器恢复以后保持一致
  "Type": "AWS::EC2::EIP",
  "DependsOn": "GatewayToInternet",
  "Properties": {
    "InstanceId": {"Ref": "Server"},
    "Domain": "vpc"
  }
},
"Server": {      <--启动一个虚拟服务器来运行Jenkins 服务器
  "Type": "AWS::EC2::Instance",
  "Properties": {
    "InstanceType": "t2.micro",      <--恢复的是t2 类型的实例
    "KeyName": {"Ref": "KeyName"},
    "UserData": {"Fn::Base64": {"Fn::Join": [ "", [      <--用户数据中包含了一个
个shell 脚本，这个脚本将在启动阶段被执行，用以在虚拟服务器上安装Jenkins 服务器
    "#!/bin/bash -ex\n",
    "wget http://pkg.jenkins-ci.org/redhat/
    ➡ jenkins-1.616-1.1.noarch.rpm\n",
    "rpm --install jenkins-1.616-1.1.noarch.rpm\n",
    [...]
    "service jenkins start\n"
  ]}}},
  [...]
}
},
"RecoveryAlarm": {      <--创建一个CloudWatch 告警来监控虚拟服务器的运行状况
  "Type": "AWS::CloudWatch::Alarm",
  "Properties": {
    "AlarmDescription": "Recover server when underlying hardware fails.",
    "Namespace": "AWS/EC2",      <--由EC2 服务提供的监控指标，使用的命名空间是A
WS/EC2
```

```

    "MetricName": "StatusCheckFailed_System",      <-- EC2 实例运行状况检查指
标名包含系统检查失败的事件
    "Statistic": "Minimum",      <-- 统计函数应用到指标。如果单个状态监测失败，
最小值将会被通知
    "Period": "60",      <-- 统计函数应用的时间，以秒为单位，必须是60 的倍数
    "EvaluationPeriods": "5",      <-- 将数据与阈值进行比较的周期数
    "ComparisonOperator": "GreaterThanThreshold",      <-- 触发告警的阈值
    "Threshold": "0",      <-- 用于将统计功能的输出与阈值进行比较的运算符
    "AlarmActions": [{      <-- 告警时执行的动作。对EC2 实例使用预先定义好的恢复
动作
        "Fn::Join": [ "", [ "arn:aws:automate:", { "Ref": "AWS::Region" },
        ":ec2:recover" ] ]
    }],
    "Dimensions": [{ "Name": "InstanceId", "Value": { "Ref": "Server" } }]
    <-- 虚拟服务器指标的一个维度
}
}
[...]
```

CloudFormation模板的创建和Jenkins在虚拟服务器上的安装需要几分钟时间。运行以下命令可以得到堆栈的结果输出。如果结果为空，过几分钟之后可以重试一下：

```

$ aws cloudformation describe-stacks --stack-name jenkins-recovery \
--query Stacks[0].Outputs
```

如果该查询结果正如这里所显示的，包含一个链接、一个用户和一个密码，那么这个堆栈创建就好了，Jenkins服务器准备好可用了。在浏览器中打开这个链接，用之前选择的admin 用户名和密码登录Jenkins服务器：

```

[
  {
    "Description": "URL to access web interface of Jenkins server.",
    "OutputKey": "JenkinsURL",
    "OutputValue": "http://54.152.240.91:8080"      <-- 在浏览器中打开这个URL
, 访问Jenkins 服务器的Web 界面
  },
  {
    "Description": "Administrator user for Jenkins.",
```

```
"OutputKey": "User",  
"OutputValue": "admin"    <---使用这个用户名登录Jenkins 服务器  
,  
{  
  "Description": "Password for Jenkins administrator user.",  
  "OutputKey": "Password",  
  "OutputValue": "*****"    <---使用这个密码登录Jenkins 服务器  
}  
]
```

现在我们可以已在Jenkins服务器上创建第一个作业。同时，我们需要用之前输出的用户名和密码进行登录。图11-3展示的是Jenkins服务器的登录表单。

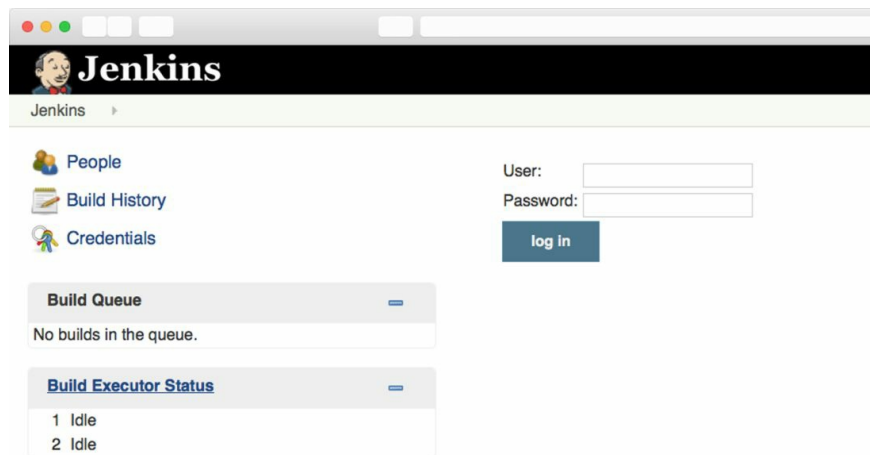


图11-3 Jenkins服务器的Web界面

Jenkins服务器运行在带自恢复功能的虚拟服务器上。如果虚拟服务器由于宿主机的问题出现故障，它将恢复所有数据和相同的IP。由于虚拟服务器用了弹性IP地址，所以链接不会改变。所有数据将恢复，因为新的虚拟服务器和之前的虚拟服务器使用一样的EBS卷。

遗憾的是，我们不能测试恢复的过程。CloudWatch告警可以监控宿主系统的运行状况，但这只能由AWS控制。

#### 资源清理

现在完成了这个例子，可以清理所有资源以避免不必要的花费。执行以下命令删除所有Jenkins配置相关的资源：

```
$ aws cloudformation delete-stack --stack-name jenkins-recovery
$ aws cloudformation describe-stacks --stack-name jenkins-recovery←--重试这条命令，直到状态
改变为DELETE_COMPLETE或者出现一个堆栈不存在的错误
```

## 11.2 从数据中心故障中恢复

如前一节描述的，底层硬件和软件失败之后，由系统状态检查和CloudWatch恢复虚拟服务器是可能的。但如果由于电力、火或者其他因素导致整个数据中心故障那会发生什么？正如11.1节描述的恢复虚拟服务器将会失效，因为那是在同一个数据中心启动EC2实例。

AWS的设计理念是“假定失败”，即便是很小的概率发生整个数据中心故障。AWS的区域是由多个数据中心组成的集群，我们把这个集群称之为可用区（Availability Zone, AZ）。结合用量去定义虚拟服务器的数量和类型来支撑AWS，必须时刻保持运行状态，在自动扩展的帮助下，你可以在数据中心在以很短的宕机时间恢复并启动虚拟服务器。另外，在多个可用区内搭建一个高可用的架构有两个注意点。

- 存储在网络附加存储的数据故障转移到另一个数据中心之后默认不可用。
- 在另一个数据中心不能用同一个私有IP地址启动新的虚拟服务器。另外，恢复后不能自动保持同一个公有IP地址，正如前一节中用CloudWatch告警触发恢复。

在本节中，我们将改善前一节中的Jenkins设置，增加整个数据中心故障恢复和解决陷阱的能力。

### 11.2.1 可用区：每个区域有多个数据中心

正如你已经了解的，AWS在全球范围内运营着多个地理位置，称为区域（region）。如果到目前为止你一直在跟进前面的示例，你已经使用了美国东部（弗吉尼亚北部）区域，也称为us-east-1。截至2018年1月，一共有18个公开可用的区域，分别位于北美洲、南美洲、欧洲和亚洲。

每个区域由多个可用区组成。一个可用区可以理解为是一组数据中心的集合，区域是由多个独立的数据中心组成，每个数据中心之间有足够的距离。可用区与可用区之间通过低延时的链路相连，所以不同可用



区之间的请求并不会像走互联网一样贵。可用区的数量取决于区域。例如，截至2018年1月美国东部（弗吉尼亚北部）区域在目前有6个可用区，欧洲（法兰克福）有3个可用区。图11-4阐述了一个区域内可用区的概念。

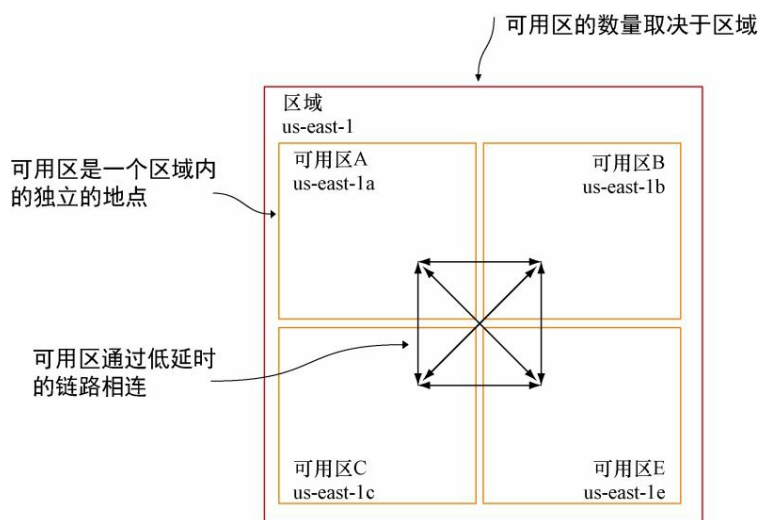


图11-4 一个区域由多个通过低延时链路相连的可用区组成

有一些AWS的服务器是高可用的，甚至默认就具有容错机制。对于有些服务，客户必须自己通过可靠的工具来搭建高可用的架构。如图11-5所示，使用多个可用区甚至多个区域来搭建一个高可用的架构也是如此。

- 有些全球性的服务跨多个区域：Route 53（DNS）和 CloudFront（CDN）。
- 有些服务在一个区域中用了多个可用区，因此可以从数据中心故障中恢复：S3（对象存储）和DynamoDB（NoSQL数据库）。
- 关系型数据库（RDS）提供了主-备设置，称为多可用区部署。如果有必要，可以将故障转移到另一个可用区。
- 虚拟服务器运行在单一可用区中。但是AWS提供了工具基于EC2搭建架构，可以从另一个可用区中故障转移。

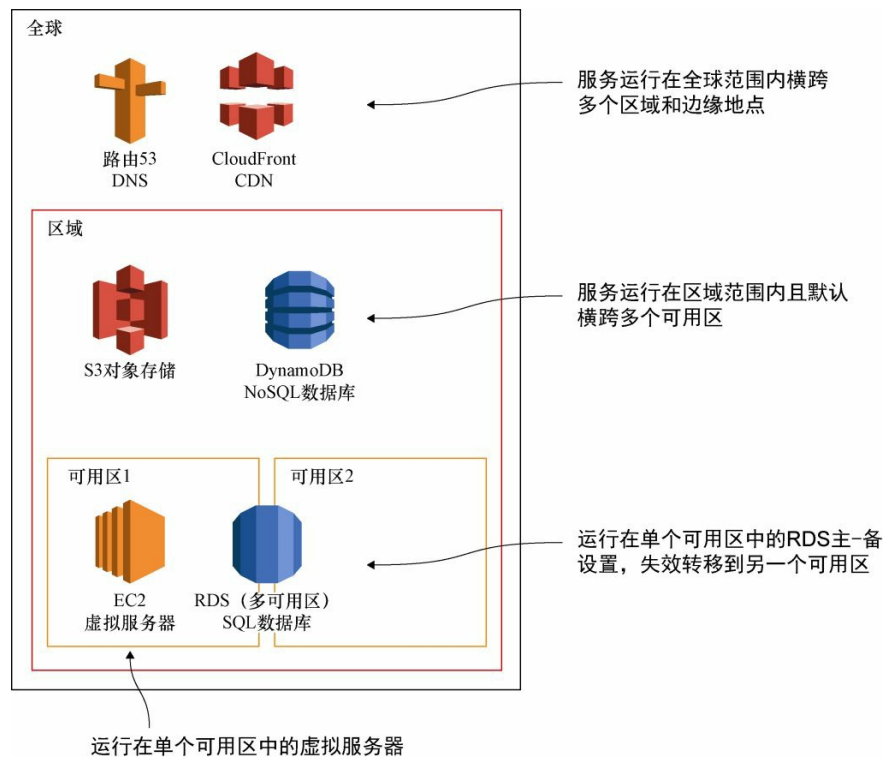


图11-5 AWS的服务可以运行在单一可用区中、跨可用区，甚至在全球范围内跨区域

可用区的标识符由区域（如us-east-1）的标识符和一个字母（a、b、c、d或者e）组成。us-east-1a是区域us-east-1内可用区的标识符。为了让资源横跨分布在不同的可用区，一个可用区的标识符针对每个AWS账户是随机生成的，这意味着在你的AWS账号中us-east-1a指向了另一个物理数据中心，在我的AWS账号中也一样。

客户可以用以下这条命令查看在自己的账号中的所有区域：

```
$ aws ec2 describe-regions
{
  "Regions": [
    {
      "Endpoint": "ec2.eu-central-1.amazonaws.com",
      "RegionName": "eu-central-1"
    },
    {
      "Endpoint": "ec2.sa-east-1.amazonaws.com",
      "RegionName": "sa-east-1"
    },
    {
      "Endpoint": "ec2.ap-northeast-1.amazonaws.com",
```

```

    "RegionName": "ap-northeast-1"
  },
  {
    "Endpoint": "ec2.eu-west-1.amazonaws.com",
    "RegionName": "eu-west-1"
  },
  {
    "Endpoint": "ec2.us-east-1.amazonaws.com",
    "RegionName": "us-east-1"
  },
  {
    "Endpoint": "ec2.us-west-1.amazonaws.com",
    "RegionName": "us-west-1"
  },
  {
    "Endpoint": "ec2.us-west-2.amazonaws.com",
    "RegionName": "us-west-2"
  },
  {
    "Endpoint": "ec2.ap-southeast-2.amazonaws.com",
    "RegionName": "ap-southeast-2"
  },
  {
    "Endpoint": "ec2.ap-southeast-1.amazonaws.com",
    "RegionName": "ap-southeast-1"
  }
]
}

```

为了列出每个区域中所有的可用区，运行以下命令，并在命令行中用**RegionName** 替换**\$Region**：

```

$ aws ec2 describe-availability-zones --region $Region
{
  "AvailabilityZones": [
    {
      "State": "available",
      "RegionName": "us-east-1",
      "Messages": [],
      "ZoneName": "us-east-1a"
    },
    {
      "State": "available",
      "RegionName": "us-east-1",

```

```
    "Messages": [],
    "ZoneName": "us-east-1b"
  },
  {
    "State": "available",
    "RegionName": "us-east-1",
    "Messages": [],
    "ZoneName": "us-east-1c"
  },
  {
    "State": "available",
    "RegionName": "us-east-1",
    "Messages": [],
    "ZoneName": "us-east-1e"
  }
]
}
```

在基于EC2实例搭建一个在多个可用区之间自动故障转移的高可用架构之前，还有一些知识是需要了解的。如果借助虚拟私有网络（VPC）在AWS内定义一个私有网络，需要了解：

- VPC总是绑定到一个区域。
- VPC内的一个子网链接到一个可用区。
- 虚拟服务器运行在单个子网中。

图11-6展示了这些依赖关系。

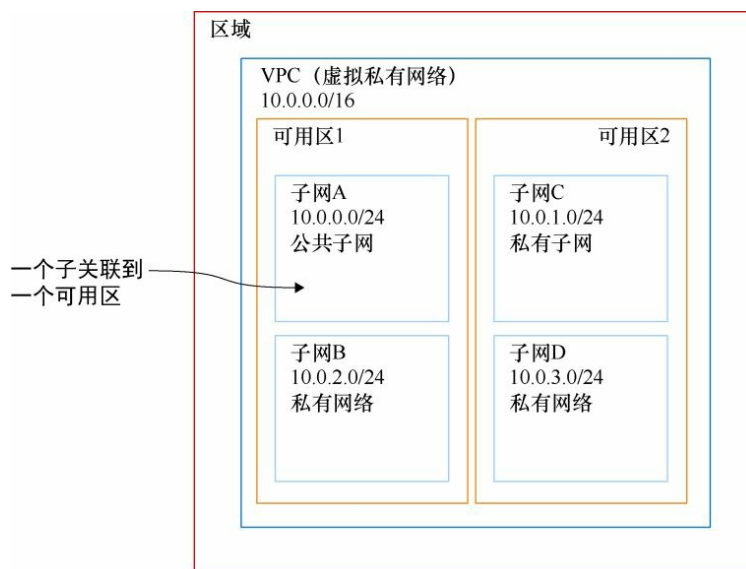


图11-6 VPC仅属于一个特定区域，子网仅和一个可用区关联

接下来，我们将会学习如何启动虚拟服务器，并且如果出现故障这个虚拟服务区可以在另一个可用区内自动重启。

## 11.2.2 使用自动扩展确保虚拟服务器一直运行

自动扩展是EC2服务的一部分，可以帮助你确保按指定数量的虚拟服务器一直运行。你可以使用自动扩展启动一个虚拟服务器，确保当原始虚拟服务器故障时可以启动新的虚拟服务器。通过自动扩展，你可以在多个子网中启动EC2实例，在整个可用区出现故障的情况下，新的虚拟服务器可以在另一个可用区的子网中启动。

配置自动扩展，需要创建配置以下两个部分。

- 启动配置 包含了虚拟服务器启动的所有信息：实例类型（虚拟服务器的大小）和启动所需的映像（AMI）。
- 自动扩展组 会告诉EC2按指定的启动项配置启动多少个虚拟服务器，如何监控实例，应该在哪个子网中启动。

图11-7展示了这个过程。

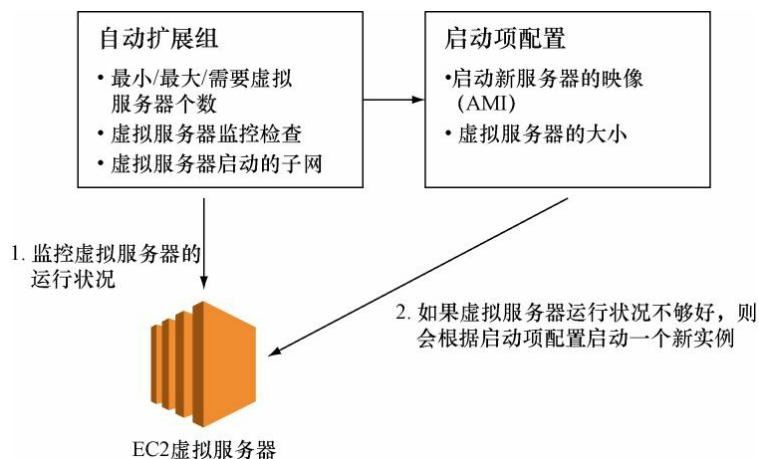


图11-7 自动扩展确保指定数量的虚拟服务器处于运行

代码清单11-3展示了如何使用自动扩展来确保单个EC2实例一直运行。表11-1展示了具体的参数。

表11-1 启动项配置和自动扩展组需要的参数

内 容	属 性	描 述	值
LaunchConfiguration	ImageId	被启动虚拟服务器的AMI的ID	账号中任何可用的AMI的ID
LaunchConfiguration	InstanceType	虚拟服务器的大小	所有可用的实例，如t2.micro、m3.medium、c3.large
AutoScalingGroup	DesiredCapacity	某一时刻所需虚拟服务器的数量	任何正整数。如果想基于启动配置启动一台虚拟服务器，那么使用1
AutoScalingGroup	MinSize	使用自动扩展允许同时运行虚拟服务器的最小值	任何正整数。如果想基于启动配置启动一台虚拟服务器，那么使用1
AutoScalingGroup	MaxSize	使用自动扩展允许同时运行虚拟服务器的最大值	任何正整数。如果想基于启动配置启动一台虚拟服务器，那么使用1

AutoScalingGroup	VPCZoneIdentifier	虚拟服务器启动所在子网的ID	账号中VPC内的任何子网ID
AutoScalingGroup	HealthCheckType	区分失效的虚拟服务器的监控检查。如果检查失败，自动扩展用新的代替	EC2使用虚拟服务器的状态检查，或者使用ELB检查负载均衡的监控状态（见第13章）

代码清单11-3 配置自动扩展组和启动配置

```
[...]
"LaunchConfiguration": {
  "Type": "AWS::AutoScaling::LaunchConfiguration",
  "Properties": {      <--用于自动扩展的启动配置
    "ImageId": "ami-1ecae776" [      <--启动虚拟服务器的映像（AMI）
    "InstanceType": "t2.micro",      <--虚拟服务器的大小
  }
},
"AutoScalingGroup": {
  "Type": "AWS::AutoScaling::AutoScalingGroup",      <--自动扩展组负责启动虚拟服务器
  "Properties": {
    "LaunchConfigurationName": {"Ref": "LaunchConfiguration"},      <--关联到启动配置
    "DesiredCapacity": 1,      <--EC2 实例的需要数量
    "MinSize": 1,      <--EC2 实例的最小数量
    "MaxSize": 1,      <--EC2 实例的最大数量
    "VPCZoneIdentifier": [      <--在子网A（在可用区A 中的）和子网B（在可用区A 中的）中启动虚拟服务器（在可用区B 中）
      {"Ref": "SubnetA"},
      {"Ref": "SubnetB"}
    ],
    "HealthCheckType": "EC2"      <--使用EC2 内部的运行状况检查
  }
}
[...]
```

自动扩展组可以根据系统的用量自动扩展虚拟服务器的数量。我们

将在第14章中学习如何根据当前的负载扩展虚拟服务器的数量。在本章中，我们只需确保一台虚拟服务器一直运行。因为需要一台虚拟服务器，为了自动扩展设置以下参数为1：

- DesiredCapacity;
- MinSize;
- MaxSize。

接下来的一节将复用本章开始的Jenkins示例，在实践中展示如何用自动扩展实现高可用性。

### 11.2.3 在另一个可用区中通过自动扩展恢复失效的虚拟服务器

本章一开始就介绍了，万一失效，可以使用CloudWatch告警触发运行着Jenkins CI服务器的虚拟服务器的恢复。在需要的情况下，这一机制会启动一个原始虚拟服务器的副本。这只是发生在同一个可用区中，因为虚拟服务器的私有IP地址和EBS卷是和一个子网和一个可用区绑定的。但是假设AWS的区域出现数据中心的故障，你的团队不会满意自己无法使用Jenkins服务器去测试、搭建和部署新软件这一事实，你需要寻找一种能够让你在另一个可用区中恢复的工具。

在自动扩展的辅助下，运行Jenkins的虚拟服务器故障转移到另一个可用区中将变为可能。这个例子的CloudFormation模板可以在下载的源代码中找到，其中multiaz.json就是我们在本章中讨论过的文件。同一文件在S3上位于<https://s3.amazonaws.com/awsinaction/chapter11/multiaz.json>。

执行以下命令创建虚拟服务器，通过自动扩展，如果需要的话，可将故障转移到另一个可用区中。用8~40个字母和数字组成的密码替换\$Password。通过以下命令用代码清单11-3所示的CloudFormation模板来设置环境：

```
$ aws cloudformation create-stack --stack-name jenkins-multiaz \
--template-url https://s3.amazonaws.com/\
awsinaction/chapter11/multiaz.json \
```



```
--parameters ParameterKey=JenkinsAdminPassword,ParameterValue=$Password
```

读者可在CloudFormation模板中按代码清单11-4找到启动配置和自动扩展组。在前一节中，当通过CloudWatch恢复告警启动单台虚拟服务器时，使用了启动配置的最重要的一些参数。

- **ImageId** ——虚拟服务器的映像（AMI）的ID。
- **InstanceType** ——虚拟服务器的大小。
- **KeyName** ——SSH密钥对的名称。
- **SecurityGroupIds** ——关联的安全组。
- **UserData** ——引导安装Jenkins CI服务器期间执行的脚本。

单个EC2实例的定义和启动配置之间有一个重要的区别：虚拟服务器的子网没有在启动配置中定义，而是在自动扩展组中定义的，如代码清单11-4所示。

代码清单11-4 在两个可用区中自动扩展的Jenkins CI服务器

```
[...]
"LaunchConfiguration": {
  "Type": "AWS::AutoScaling::LaunchConfiguration",      <-- 用于自动扩展的启动配置
  "Properties": {
    "InstanceMonitoring": false,      <-- 默认情况下，EC2 每5 min发送指标到CloudWatch，当然你也可以通过额外付费，启动更详细的实例监控，每分钟获得指标
    "ImageId": {"Fn::FindInMap": [      <-- 启动虚拟服务器的映像（AMI）
      "EC2RegionMap",
      {"Ref": "AWS::Region"},
      "AmazonLinuxAMIHVMESBBacked64bit"
    ]},
    "KeyName": {"Ref": "KeyName"},      <-- 远程SSH 登录到虚拟服务器的密钥
    "SecurityGroups": [{"Ref": "SecurityGroupJenkins"}],      <-- 附加到虚拟服务器的安全组
    "AssociatePublicIpAddress": true,      <-- 给虚拟服务器启动公有IP 地址
    "InstanceType": "t2.micro",      <-- 虚拟服务器的类型
    "UserData": {      <-- 用户数据包含一个脚本，这个脚本是在虚拟服务器里安装Jenkins 服务器，在虚拟服务器引导配置时执行
      "Fn::Base64": {
        "Fn::Join": [
          "",
          [
```

```

        "#!/bin/bash -ex\n",
        "wget http://pkg.jenkins-ci.org/redhat/
        ➡ jenkins-1.616-1.1.noarch.rpm\n",
        "rpm --install jenkins-1.616-1.1.noarch.rpm\n",
        [...]\n"
    "service jenkins start\n"
  ]
]
}
}
}
},
"AutoScalingGroup": {
  "Type": "AWS::AutoScaling::AutoScalingGroup",      ←--自动扩展组负责启动虚
拟服务器
  "Properties": {
    "LaunchConfigurationName": {"Ref": "LaunchConfiguration"},      ←--关联
到启动配置
    "Tags": [      ←--自动扩展组的标签
      {
        "Key": "Name",
        "Value": "jenkins",
        "PropagateAtLaunch": true      ←--将相同的标签附加到由此自动扩展组启动
的虚拟服务器
      }
    ],
    "DesiredCapacity": 1,      ←--EC2 实例的需要个数
    "MinSize": 1,      ←--EC2 实例的最小个数
    "MaxSize": 1,      ←--EC2 实例的最大个数
    "VPCZoneIdentifier": [      ←--在子网A（创建在可用区A）和子网B（创建在可用
区B）中启动虚拟服务器
      {"Ref": "SubnetA"},
      {"Ref": "SubnetB"}
    ],
    "HealthCheckType": "EC2"      ←--使用EC2 服务内部的运行状况检查
  }
}
[...]
```

CloudFormation模板的创建需要几分钟。执行以下命令获得虚拟服务器的公有IP地址。如果没有IP地址出现，说明虚拟服务器还没启动完成，可以过一会儿再试。

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
--query "Reservations[0].Instances[0].\
[InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-e8c2063b",      <-- 虚拟服务器的实例ID
  "52.4.11.10",      <-- 虚拟服务器的公有IP 地址
  "10.0.1.56",       <-- 虚拟服务器的私有IP 地址
  "subnet-36257a41"  <-- 虚拟服务器的子网ID
]
```

在浏览器中打开[http://\\$PublicIP:8080](http://$PublicIP:8080)，用之前的**describe** 命令输出的公有IP地址替换**\$PublicIP**，Jenkins服务器的Web界面就出现了。

执行以下命令终止虚拟服务器，测试自动扩展的恢复过程。用之前**describe** 命令输出的实例ID替换**\$InstanceId**：

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

几分钟之后自动扩展组检测到虚拟服务器被终止了，然后启动一台新的虚拟服务器。重新运行**describe-instances** 命令，直到出现一台运行的虚拟服务器信息。

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
--query "Reservations[0].Instances[0].\
[InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-5e4f68f7",
  "54.88.118.96",
  "10.0.0.36",
  "subnet-aa29b281"
]
```

对于新实例，实例ID、公有IP地址、私有IP地址甚至子网ID都变了。在浏览器中打开[http://\\$PublicIP:8080](http://$PublicIP:8080)，用之前的**describe** 命令输

出的公有IP地址替换\$PublicIP，Jenkins服务器的Web界面就出现了。

通过自动扩展，你已经搭建了一个由EC2组成的高可用的架构。当前的步骤中存在下面两个问题。

- Jenkins服务器的数据存储在磁盘上，当出现故障时，一台新的虚拟服务器被启动，新的磁盘会被创建，数据将丢失。
- 新的虚拟服务器恢复时，Jenkins服务器的公有IP地址、私有IP地址发生了改变。Jenkins服务器在同一个端点下变为不可用。

接下来我们将学习如何解决这些问题。

## 11.2.4 陷阱：网络附加存储恢复

EBS服务为虚拟服务器提供了网络附加存储（NAS）。EC2关联到一个子网，子网关联到一个可用区。EBS卷存在于单个可用区中。如果虚拟服务器由于故障在另一个可用区中启动，存储在EBS卷中的数据将不再可用。图11-8阐述了这个问题。

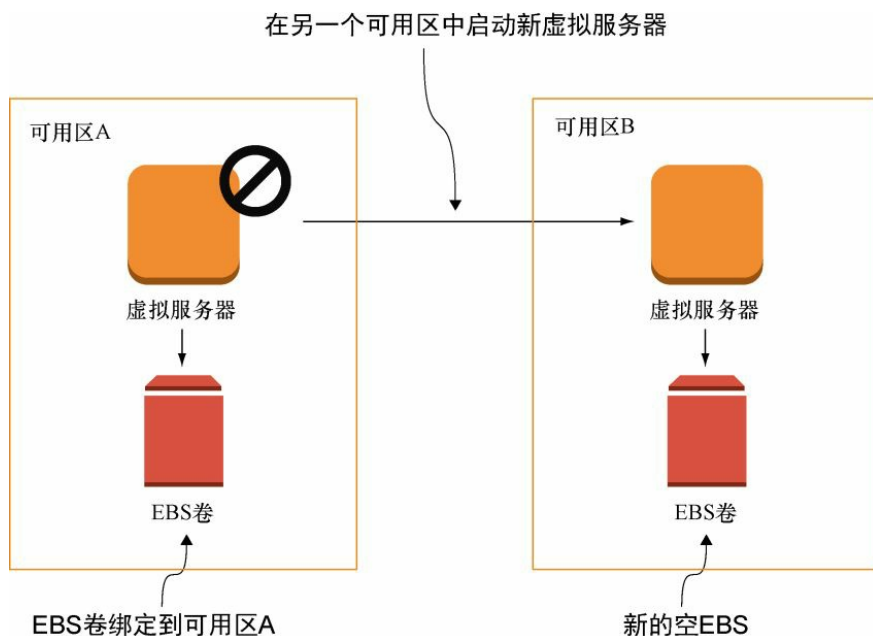


图11-8 EBS卷只有在单个可用区中可用

这个问题有很多种解决方案。

- 将虚拟服务器的状态移到托管的服务里，默认使用多个可用区：RDS（关系型数据库）、DynamoDB（NoSQL数据库）或者S3（对象存储）。
- 给EBS卷创建快照，如果虚拟服务器需要在另一个可用区中恢复，则可使用快照。EBS快照存储在S3中以便在多个可用区中保持可用。
- 使用分布式的第三方存储解决方案（GlusterFS、DRBD、MongoDB等），在多个可用区中存储数据。

Jenkins服务器直接在磁盘上存储数据。为了在外面存放虚拟服务器的状态，不能使用RDS、DynamoDB或者S3，需要用快照级别的存储解决方案代替。正如我们所学到的，EBS卷只有单个可用区中可用，所以这不能很好地解决这个问题。使用分布式第三方存储解决方案可行，但也引入了很多超出了本书范围的复杂性。我们将学习如何使用EBS快照在另一个可用区中恢复一台虚拟服务器，并且没有丢失存储在EBS卷里面的完整状态数据。

如代码清单11-5所示，在启动配置的辅助下可以为通过自动扩展启动的虚拟服务器指定一个自己的映像（AMI），映像就类似于EBS快照，包含操作系统虚拟化相关的其他信息。客户可以基于这个AMI启动一个新的虚拟服务器，但不能用EBS快照创建一个根卷。客户创建任何运行的虚拟服务器的映像（AMI）。与EBS卷本身相比，EBS快照和映像AMI是存储在一个区域内的多个可用区的，所以客户可以通过它在另一个可用区恢复。

代码清单11-5 更新映像，在失败时启动新的虚拟服务器

```
[...]
"LaunchConfiguration": {
  "Type": "AWS::AutoScaling::LaunchConfiguration",
  "Properties": {
    "InstanceMonitoring": false,
    "ImageId": {"Ref": "AMISnapshot"},      <-- 自动扩展根据指定的AMI启动新的虚
拟服务器
    "KeyName": {"Ref": "KeyName"},
    "SecurityGroups": [{"Ref": "SecurityGroupJenkins"}],
    "AssociatePublicIpAddress": true,
    "InstanceType": "t2.micro",
    "UserData": {
      "Fn::Base64": {
        "Fn::Join": [
```

```
    "",
    [
        "#!/bin/bash -ex\n",
        "wget http://pkg.jenkins-ci.org/redhat/
        ➡jenkins-1.616-1.1.noarch.rpm\n",
        "rpm --install jenkins-1.616-1.1.noarch.rpm\n",
        [...]\n",
        "service jenkins start\n"
    ]
  ]
}
}
}
}
[...]
```

我们将执行以下步骤。

- (1) 为Jenkins CI服务器增加一个作业。
- (2) 用虚拟服务器的当前状态的快照创建一个AMI。
- (3) 更新启动配置。
- (4) 测试恢复情况。

执行以下命令来获得正在运行的虚拟服务器的实例ID和公有IP地址：

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
--query "Reservations[0].Instances[0].[InstanceId, PublicIpAddress]"
[
    "i-5e4f68f7",
    "54.88.118.96"
]
```

现在，通过以下步骤创建一个新的Jenkins作业。

(1) 在浏览器中打开[http://\\$PublicIP:8080/newJob](http://$PublicIP:8080/newJob)，用之前的 **describe** 命令输出的公有IP地址替换**\$PublicIP**。

(2) 用启动CloudFormation模板时选择的**admin** 用户名和密码进行登录。

(3) 输入**AWS in Action** 作为新作业的名字。

(4) 选择“Freestyle Project”（自由式项目）作为作业类型，点击“OK”按钮保存作业。

我们已经对存储在EBS根卷里的虚拟服务器的状态做了一些修改。

在失效的情况下，为了确保通过自动扩展组启动虚拟服务器的新作业不会丢失，需要创建一个AMI作为当前状态的快照。执行以下命令来实现这一点，将**\$InstanceId** 替换为前面**describe** 命令中的实例ID。

```
$ aws ec2 create-image --instance-id $InstanceId --name jenkins-multiaz \
{
  "ImageId": "ami-0dba4266"      <---利用CloudFormation 更新启动配置里的新AMI
  的ID
}
```

等到映像变为可用的，执行以下命令检查当前状态，用**create-image** 命令输出的**ImageId** 替换**\$ImageId**：

```
$ aws ec2 describe-images --image-id $ImageId --query "Images[].State"
```

我们需要通过代码清单11-5所示的CloudFormation 模板更新启动配置。执行以下命令来实现这一点，用**ImageId** 替换**\$ImageId**：

```
$ aws cloudformation update-stack --stack-name jenkins-multiaz \
--template-url https://s3.amazonaws.com/awsinaction/\
chapter11/multiaz-ebs.json --parameters \
ParameterKey=JenkinsAdminPassword,UsePreviousValue=true \
ParameterKey=AMISnapshot,ParameterValue=$ImageId
```

等待几分钟，直到CloudFormation已经更换启动配置。运行`aws cloudformation describe-stacks--stack-name jenkins-multiaz` 来检查状态，直到状态变更为UPDATE\_COMPLETE。现在模拟虚拟服务器的工作，执行以下命令终止虚拟服务器，用`describe` 命令输出的结果替换`$InstanceId`：

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

自动扩展组检测丢失的虚拟服务器，并开启一个新的虚拟服务器需要5 min。运行以下命令得到新启动虚拟服务器的信息。如果输出是空值，几分钟后重试命令执行：

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
--query "Reservations[0].Instances[0].[InstanceId, PublicIpAddress]"
```

在浏览器中打开`http://$PublicIP:8080`，用之前的`describe` 命令输出的公有IP地址替换`$PublicIP`，就会在Jenkins Web接口中看到可执行作业的名字。

## 资源清理

为了避免不必要的花费，接下来清理一下资源。执行以下命令准备删除未使用的资源：

```
$ aws ec2 describe-images --owners self \
--query Images[0].[ImageId,BlockDeviceMappings[0]\
.Ebs.SnapshotId]
```

输出内容包含映像（AMI）的ID，以及相应快照的ID。执行以下命令删除相应Jenkins设置的所有资源。用之前输出的映像ID替换`$ImageId`，快照ID替换`$SnapshotId`。

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz
$ aws cloudformation describe-stacks --stack-name jenkins-multiaz    <---重复执行命令直到状态变更为DELETE_COMPLETE 或者出现堆栈不存在的错误
$ aws ec2 deregister-image --image-id $ImageId
$ aws ec2 delete-snapshot --snapshot-id $SnapshotId
```



## 11.2.5 陷阱：网络接口恢复

正如本章一开始描述的，在同一个可用区内，通过CloudWatch告警的辅助来恢复虚拟服务器，可以很容易地保持私有IP地址和公有IP地址不变。客户可以在故障转移后使用这个IP地址作为一个端点去访问服务。

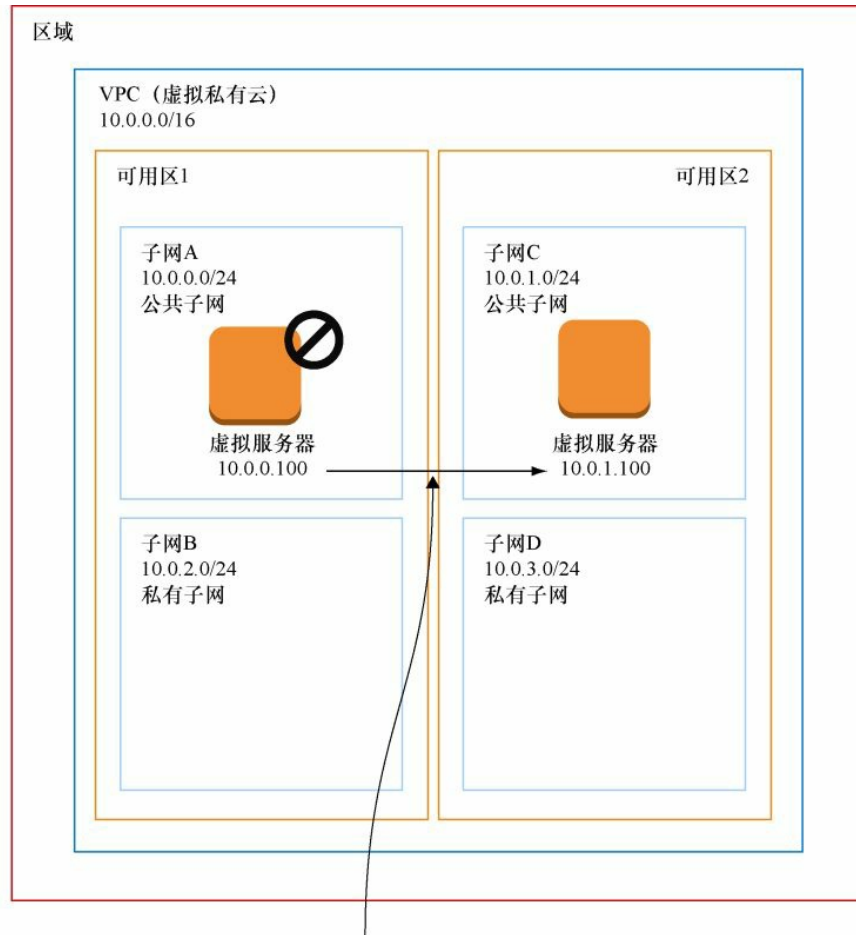
当使用自动扩展从服务器或数据中心故障恢复时不能这样做。如果一个虚拟服务器必须在另一个可用区内从数据中心故障中恢复，它必须在另一个子网内启动。这时虚拟服务器不能使用相同的私有IP地址，如图11-9所示。

默认情况下，通过自动扩展启动的虚拟服务器不能把弹性IP作为公有IP。但其实用一个静态端点去接收请求，这种需求是常见的。对于Jenkins服务器的使用场景，开发者想用IP地址或者主机名访问Web接口。当通过使用自动扩展功能为单一虚拟服务器构建高可用性时，有多种不同的情况可能用来提供一个静态端点。

- 分配弹性IP，在虚拟服务器引导程序中绑定这个公有IP地址。
- 创建或者更新一个DNS条目关联到当前虚拟服务器的公共或私有IP地址。
- 使用弹性负载均衡器（ELB）作为一个静态端点，将请求路由到当前虚拟服务器。

使用第二种方案需要用Route53（DNS）服务关联一个域名。目前已经跳过这种方案，因为需要注册一个域名来实现。ELB（弹性负载均衡）方案将在第12章中介绍，本节先跳过不讲。

因此，我们关注于第一种方案：通过自动扩展在虚拟服务器引导程序中分配一个弹性IP，并且关联这个公有IP地址。



由于虚拟服务器是从另一个子网中恢复的，所以私有IP地址改变了

图11-9 在出现故障时，虚拟服务器在另一个子网中启动，私有IP地址改变

基于自动扩展，再次执行以下命令创建Jenkins配置，使用弹性IP作为一个静态端点。

```
$ aws cloudformation create-stack --stack-name jenkins-elasticip \
--template-url https://s3.amazonaws.com/\
awsinaction/chapter11/multi-az-elasticip.json \
--parameters ParameterKey=JenkinsAdminPassword,ParameterValue=$Password \
--capabilities CAPABILITY_IAM
```

在代码清单11-6所示模板的基础上，该命令创建堆栈。通过自动扩展运行Jenkins服务器，与原始模板不同之处如下。

- 分配弹性IP。

- 在用户数据的脚本中，加上关联弹性IP的命令。
- 创建IAM角色和策略，运行EC2实例来关联弹性IP。

代码清单11-6 使用弹性IP作为一个静态端点

```
[...]
"IamRole": {      <--创建一个用于EC2 实例的IAM 角色
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {"Service": ["ec2.amazonaws.com"]}
        },
        "Action": ["sts:AssumeRole"]
      ]
    },
    "Path": "/",
    "Policies": [
      {
        "PolicyName": "root",
        "PolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
            {
              "Action": ["ec2:AssociateAddress"],      <--使用此IAM 角色的EC2
              "Resource": ["*"],
              "Effect": "Allow"
            }
          ]
        }
      }
    ]
  }
},
"IamInstanceProfile": {
  "Type": "AWS::IAM::InstanceProfile",
  "Properties": {
    "Path": "/",
    "Roles": [{"Ref": "IamRole"}]
  }
},
```

实例允许关联弹性IP

```

"ElasticIP": {      <--为运行Jenkins 的虚拟服务器分配弹性IP
  "Type": "AWS::EC2::EIP",
  "Properties": {
    "Domain": "vpc"      <--为VPC 创建弹性IP
  }
},
"LaunchConfiguration": {
  "Type": "AWS::AutoScaling::LaunchConfiguration",
  "DependsOn": "ElasticIP",      <--等待直至弹性IP 可用
  "Properties": {
    "InstanceMonitoring": false,
    "IamInstanceProfile": {"Ref": "IamInstanceProfile"},
    "ImageId": {"Fn::FindInMap": [
      "EC2RegionMap",
      {"Ref": "AWS::Region"},      <--将AWS CLI 的默认区域设置为虚拟服务器正在
运行的区域
    ],
    "AmazonLinuxAMIHVMESBBacked64bit"
  ]},
  "KeyName": {"Ref": "KeyName"},
  "SecurityGroups": [{"Ref": "SecurityGroupJenkins"}],
  "AssociatePublicIpAddress": true,
  "InstanceType": "t2.micro",
  "UserData": {
    "Fn::Base64": {
      "Fn::Join": [
        "",
        [
          "#!/bin/bash -ex\n",
          "aws configure set default.region ", {"Ref": "AWS::Region"},"",
          <--从实例元数据获得实例ID
          "aws ec2 associate-address --instance-id ",      <--给虚拟服务器
关联弹性IP
          "$INSTANCE_ID --allocation-id ",
          {"Fn::GetAtt": ["ElasticIP", "AllocationId"]},
          "\n",
          "wget http://pkg.jenkins-ci.org/redhat/
          ➡jenkins-1.616-1.1.noarch.rpm\n",
          "rpm --install jenkins-1.616-1.1.noarch.rpm\n",
          [...]
          "service jenkins start\n"
        ]
      ]
    }
  }
}
}
}
}
[...]
```

如果该查询返回的输出包括URL、用户和密码，就表明这个堆栈创建好了，并且Jenkins服务器也可以使用了。在浏览器中打开这个URL，用选择的admin 用户和密码登录Jenkins服务器。如果输出是空的，几分钟后重试：

```
$ aws cloudformation describe-stacks --stack-name jenkins-elasticip \
--query Stacks[0].Outputs
```

现在可以测试虚拟服务器是否按期望恢复。接下来，我们需要知道运行虚拟服务器的实例ID。运行以下命令获取这个信息：

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
Values=jenkins-elasticip" "Name=instance-state-code,Values=16" \
--query "Reservations[0].Instances[0].InstanceId" --output text
```

执行以下命令终止虚拟服务器，通过自动扩展触发测试恢复过程。用之前的命令输出的信息替换\$InstanceId：


```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

等待几分钟恢复虚拟服务器。因为我们是通过启动时引导配置把弹性IP分配给新虚拟服务器的，所以我们可以在浏览器中打开同一个URL，这个URL就是之前终止旧实例的URL。

#### 资源清理

清理资源避免额外花费。执行以下命令删除Jenkins设置相关的所有资源：

```
$ aws cloudformation delete-stack --stack-name jenkins-elasticip
$ aws cloudformation describe-stacks --stack-name jenkins-elasticip    <--- 重新运行这个命令
，直到状态变为DELETE_COMPLETE，或者发送错误说堆栈不存在
```



现在即使运行中的虚拟服务器需要被一个可用区的另一个虚拟服务器代替，运行Jenkins的虚拟服务器的公有IP地址不会改变了。

## 11.3 分析灾难恢复的需求

在AWS上实现高可用或者容错的架构之前，你应该先分析灾难恢复需求。与传统数据中心相比，云上的灾难恢复更容易、更经济。但是这也增加了系统的复杂性，进而增加了系统的初始成本和运营成本。从业务的角度上看，对系统进行灾难恢复，恢复时间目标（RTO）和恢复点目标（RPO）的标准定义是非常重要的系统容灾的标准。

恢复时间目标（Recovery Time Objective，RTO）是让系统从失败中恢复的时间。时间的长度直到故障后达到系统服务级别。在Jenkins服务器的例子中，RTO应该是在虚拟服务器或者整个数据中心故障后，一直到新的虚拟服务器被启动，Jenkins服务器被安装并运行。

恢复点目标（Recovery Point Objective，RPO）是由失败导致的可接受数据丢失的时间点。丢失的数据量在时间内可被衡量。如果故障发生在早上10点，系统从数据快照在早上9点开始恢复，数据丢失的时间跨度是1 h。在使用自动扩展的Jenkins服务器的例子中，两个EBS快照是RPO的最大时间跨度。在另一个数据中心恢复时，Jenkins作业的配置和结果发送改变，数据在最后一个EBS快照之后将丢失。图11-10说明了RTO和RPO的定义。

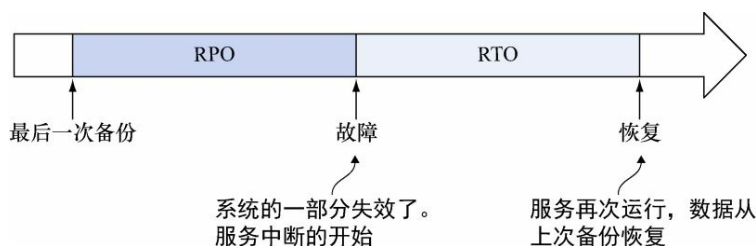


图11-10 RTO和RPO的定义

## 单个虚拟服务器的RTO和RPO的比较

要使单个服务具有高可用性，我们将学习两种可行的解决方案。表11-2对这两种解决方案进行了对比。

表11-2 对单个虚拟服务器的高可用性对比

	RTO	RPO	可用性
通过CloudWatch告警触发恢复	大约10 min	没有数据丢失	从虚拟服务器失效中恢复，但并不能从整个可用区故障中恢复
通过自动扩展恢复	大约10 min	自最后一次快照的所有数据丢失。快照的时间花费在30 min到24 h	从虚拟服务器失效中恢复，并且从整个可用区故障中恢复

如果想在可用区之外恢复并降低RPO，应该尝试实现无状态服务器。使用存储服务，如RDS、S3和DynamoDB服务，可以做到这一点。如果需要使用这些服务，可阅读本书的第三部分。



## 11.4 小结

- 如果底层硬件或者软件出现故障，虚拟服务器将失效。
- 可以借助CloudWatch告警的帮助来恢复已经失效的虚拟服务机。
- AWS区域由多个独立的数据中心组成，这些数据中心称之为可用区。
- 使用多可用区部署可从数据中心故障中恢复。
- 虽然有一些服务默认使用多可用区部署，但虚拟服务器是运行在单个可用区内的。
- 如果一个可用区失效了，可以使用自动扩展来保证单个虚拟服务器总处于运行状态。
- 当数据存储于EBS卷中，而不是使用RDS、S3和DynamoDB这样的托管服务时，在另一个可用区中恢复数据是不太现实的。

## 第12章 基础设施解耦：ELB与SQS

### 本章主要内容

- 系统解耦的原因
- 利用负载均衡器同步解耦
- 利用消息队列异步解耦

设想一下你打算从我这里得到一些关于使用AWS的建议，因此我们计划在咖啡馆见个面。为了使这次会面成功，我们必须具备这样几个条件：

- 同时有空；
- 在同一个地点；
- 在咖啡馆找到彼此。

这次会面的问题是它与一个具体的位置密切相关。我们可以通过将会面与具体位置脱钩来解决问题，于是我们更改计划并安排使用Google Hangout来对话。那么，我们现在就必须要做到：

- 同时有空；
- 在Google Hangout上找到对方。

Google Hangout（这也适用于所有其他视频/语音聊天工具）让我们实现了同步解耦。它消除了在同一地点的要求，但仍然要求我们在同一时间对话。

我们甚至还可以通过使用电子邮件来进行沟通，摆脱时间上的束缚。现在我们可以这么做：

- 通过邮件找到彼此。

电子邮件可以做到异步解耦。我们可以在收件人睡觉的时候发出电子邮件，当他们醒来时会做出回应。

#### 示例都包含在免费套餐中

本章中的所有示例都包含在免费套餐中。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

#### 注意

要完全理解本章的内容，读者需要阅读并理解第11章中介绍的自动扩展的概念。

会面不是唯一需要解耦的事情。在软件系统中用户可以找到很多紧耦合的组件。

- 公有IP地址就像我们会面的地点一样。要向Web服务器发出请求，就必须知道对方的公有IP地址，并且必须有一台服务器与该地址相连。如果要更改公有IP地址，双方都要参与进来做适当的更改。
- 如果要向Web服务器发出请求，则Web服务器必须同时处于联机状态，否则请求将会失败。导致Web服务器离线的原因有很多，如正在安装更新、硬件故障等。

AWS为这两个问题提供了一个解决方案。弹性负载均衡（Elastic Load Balancing, ELB）服务提供了一个位于Web服务器和互联网之间的负载均衡器，可用以同步解耦服务器。对于异步解耦，AWS提供了一个简单消息队列服务（Simple Queue Service, SQS）。它提供了一个消息队列的基础设施。本章将介绍这两种服务。我们现在就从ELB开始学习吧。

## 12.1 利用负载均衡器实现同步解耦

将单个Web服务器暴露给外界会引入依赖关系，这就是EC2实例的公有IP地址。从这一点来看，不能再一次改变这个公有IP地址，因为许多客户端正用这个地址发送请求到我们的服务器。于是我们遇到了以下的问题。

- 改变公有IP地址是不可能的，因为有许多客户端依赖着它。
- 如果添加额外的服务器（以及IP地址）来处理增加的负载，则所有当前的客户端都将会忽略掉这个变化：它们仍将所有的请求发送到第一个服务器的公有IP地址。

我们可以使用指向自己的服务器的DNS名字来解决这些问题，但DNS并不完全在我们的控制之下。DNS服务器会缓存条目，有时它们不遵循我们的生存时间（Time To Live，TTL）设置。更好的解决方案就是使用负载均衡器。

负载均衡器可以帮助解耦请求者等待即时响应这一类的系统。客户不必将Web服务器暴露给外部世界，只需要将负载均衡器暴露给外界即可。然后，负载均衡器将请求重定向到其后面的Web服务器上。图12-1展示了负载均衡器是如何工作的。

AWS通过ELB服务提供负载均衡器。AWS负载均衡器具有容错和可扩展的特性。对于每个ELB，客户需要支付0.025美元/h的费用，而每GB处理流量则需支付0.008美元。这个价格适用于弗吉尼亚北部（us-east-1）区域。

### 注意

ELB服务没有独立的管理控制台，它被集成在EC2服务中。

负载均衡器可以与多个Web服务器一起使用——客户可以在处理请求/响应类型的任何系统的前端使用负载均衡器。

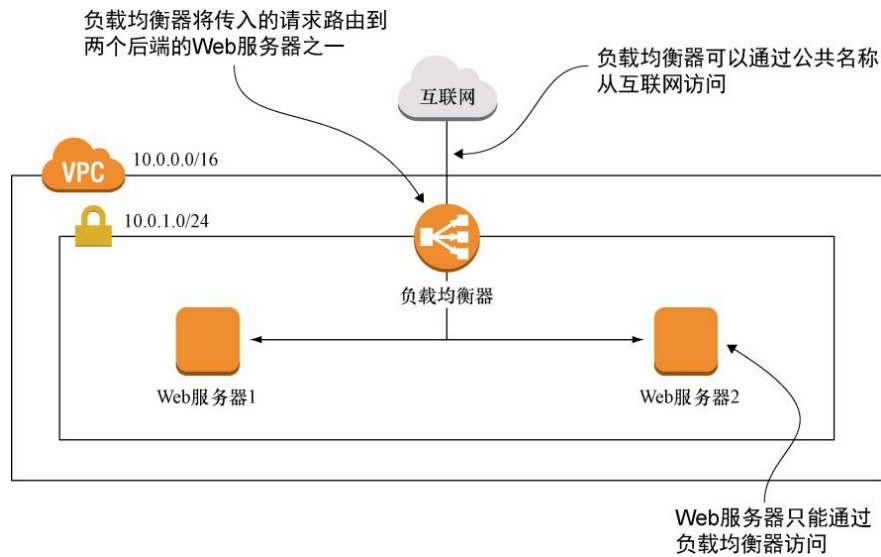


图12-1 负载均衡器同步解耦你的服务器

## 12.1.1 使用虚拟服务器设置负载均衡器

当涉及许多AWS服务集成在一起时，AWS的优势就会显现出来。在第11章中，我们了解了自动扩展组。你现在将弹性负载均衡器（ELB）放在自动扩展组之前，以便将流量与Web服务器解耦。自动扩展组将确保你始终有两台服务器正在运行。在自动扩展组中启动的服务器将自动向ELB注册。图12-2展示了设置的方式。有趣的是，Web服务器不能直接从互联网访问。只有负载均衡器是可以访问的，并将请求重定向到其后端的服务器上，这是由安全组完成的，读者可以在第6章中了解安全组的知识。

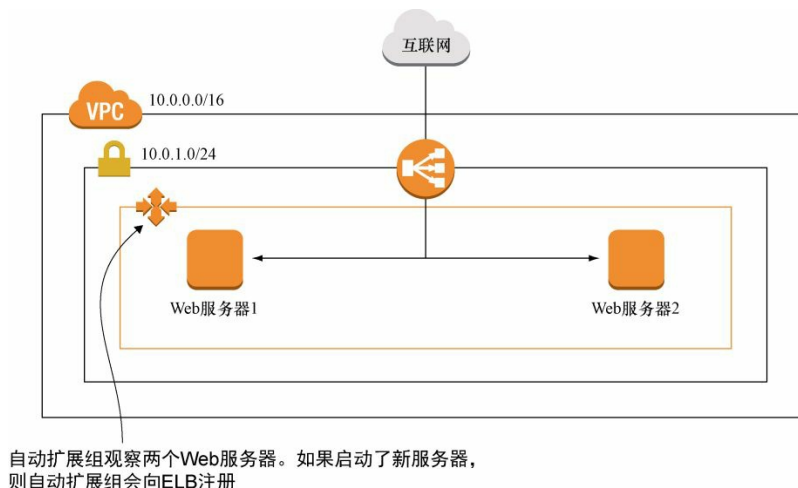


图12-2 自动扩展组与ELB紧密合作：在负载均衡器上注册新的服务

关于ELB有如下的描述。

- 将ELB附加到子网上，子网的数量可以有多个。
- ELB的端口映射到位于其后端的服务器的端口上。
- 需要分配安全组给ELB，客户可以使用与EC2实例相同的方式限制ELB的流量。
- 可以选择ELB是否可以从公网访问。

通过在自动扩展组描述中指定**LoadBalancerNames** 实现ELB和自动扩展组之间的连接。

代码清单12-1展示了一个CloudFormation模板片段，其作用是创建ELB并将其与自动扩展组连接。这个代码清单实现了图12-2所示的示例。

代码清单12-1 创建负载均衡器并将其与自动扩展组连接

```
[...]
"LoadBalancerSecurityGroup": {
  "Type": "AWS::EC2::SecurityGroup",
  "Properties": {
    "GroupDescription": "elb-sg",
    "VpcId": {"Ref": "VPC"},
    "SecurityGroupIngress": [{      <-- 负载均衡器只接受80 端口的流量
      "CidrIp": "0.0.0.0/0",
      "FromPort": 80,
      "ToPort": 80,
      "IpProtocol": "tcp"
    }]
  }
},
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    "Subnets": [{"Ref": "Subnet"}],    <-- 将ELB 附加到子网上
    "LoadBalancerName": "elb",
    "Listeners": [{      <-- 映射负载均衡器的端口到其后端的服务器端口
      "InstancePort": "80",
      "InstanceProtocol": "HTTP",
      "LoadBalancerPort": "80",
      "Protocol": "HTTP"
    }],
  },
}
```

```

    "SecurityGroups": [{ "Ref": "LoadBalancerSecurityGroup" }],      ←--分配一个安全组
    "Scheme": "internet-facing"      ←--ELB 是公开可访问的（仅用于内部而不是互联网，可以将负载均衡器定义为仅可从私有网络访问）
  },
  "LaunchConfiguration": {
    "Type": "AWS::AutoScaling::LaunchConfiguration",
    "Properties": {
      [...]
    }
  },
  "AutoScalingGroup": {
    "Type": "AWS::AutoScaling::AutoScalingGroup",
    "Properties": {
      "LoadBalancerNames": [{ "Ref": "LoadBalancer" }],      ←--将自动扩展组连接到ELB
      "LaunchConfigurationName": { "Ref": "LaunchConfiguration" },
      "MinSize": "2",
      "MaxSize": "2",
      "DesiredCapacity": "2",      ←--最小尺寸（MinSize）、最大尺寸（MaxSize）和最大容量（DesiredCapacity）的设定
      "VPCZoneIdentifier": [{ "Ref": "Subnet" }]
    }
  }
}

```

为了更好地理解ELB，我们创建了一个CloudFormation模板，这个模板位于<https://s3.amazonaws.com/awsinaction/chapter12/loadbalancer.json>。根据该模板创建一个堆栈，然后使用浏览器访问堆栈的URL输出。每次重新加载页面时，都应该可以看到后端Web服务器的私有IP地址之一。

#### 资源清理

删除创建的堆栈。

## 12.1.2 陷阱：过早地连接到服务器

自动扩展组负责将新启动的EC2实例与负载均衡器连接起来。但

是，自动扩展组如何知道EC2实例何时已经安装并准备好接受流量？遗憾的是，自动扩展组其实并不知道服务器是否准备就绪，它会在实例启动后立即向负载均衡器注册EC2实例。如果将流量发送到已启动但未就绪的服务器，则请求将失败，你的用户将会感到很不满意。

但是，ELB可以对连接的每个服务器定期进行运行状况检查，以确定服务器是否可以提供请求。在Web服务器示例中，需要检查是否获取特定资源（如/index.html）的状态响应代码200。代码清单12-2展示了如何使用CloudFormation完成此操作。

代码清单12-2 ELB健康检查以确定服务器是否能够响应请求

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    [...]
    "HealthCheck": {
      "Target": "HTTP:80/index.html",      <---服务器对/index.html 返回的状态
      代码是200 吗
      "Interval": "10",                  <---每10 s 进行一次检查
      "Timeout": "5",                   <---超时时间为5 s（必须小于Interval）
      "HealthyThreshold": "3",          <---检查连续通过3 次才能认为是运行状况良好的
      "UnhealthyThreshold": "2"         <---检查连续失败两次，则认为是运行状况不好
    }
  }
}
```

如果不去检查/index.html，还可以去请求一个动态页面，如/healthy.php，来进行一些额外的检查，以确定Web服务器是否准备好处理请求。协议中的约定就是，当服务器准备就绪时，必须返回200作为HTTP状态码。如此而已。

#### 过于激进的运行状况检查会导致服务器宕机

如果服务器过于忙，以至于无法接收运行状况检查，则ELB将停止向该服务器转发流量。如果这种情况是对你的系统只是由于常规的负载增加而引起的，ELB的反应将使情况变得更糟！我们已经看到应用程序由于过于激进的运行状况检查而遭遇停机。你需要的是合理的负载测试用来了解究竟发生了什么。一个适用的解决方案一定是针对特定的应用程序的，不可能是通用的方案。



默认情况下，自动扩展组会根据EC2每分钟执行的运行状况检查的结果来判定EC2实例是否正常。你也可以将自动扩展组配置为使用负载均衡器所运行的状况检查。不仅仅是当硬件出现故障，而且对于应用程序发生故障自动扩展组都将终止服务器。具体的做法是在自动扩展组描述中设置"**HealthCheckType**": "**ELB**"。许多时候这个设置是有意义的，因为重新启动服务器可以解决内存、线程池或磁盘空间不足等问题。但是，在应用程序已经损坏的情况下这也可能导致完全不必要的EC2实例重启。

## 12.1.3 更多使用场景

到目前为止，我们已经看到了ELB最常见的使用场景：通过HTTP将传入的Web请求负载均衡到Web服务器上。如前所述，ELB实际上可以做得更多。在本节中，我们将看看另外4个典型的使用场景。

(1) ELB能够均衡TCP流量，几乎可以将任何应用程序部署在负载均衡器的后端。

(2) 如果将SSL证书添加到AWS上，ELB可以将SSL加密过的流量转换为普通的流量。

(3) ELB可以记录下每一个请求，并将请求日志存储在S3上。

(4) ELB可以在多个可用区（AZ）之间均匀分配客户的请求。

### 1. 处理TCP流量

到目前为止，你只使用ELB来处理HTTP流量。你还可以配置ELB用来重定向纯TCP的通信，解耦使用专有接口的数据库或传统应用。与处理HTTP流量的ELB配置相比，你必须更改侦听器和健康检查的设定以实现使用ELB处理TCP流量。这种情况下，运行状况检查就不同于处理HTTP时一样检查特定的响应。当ELB打开套接字时，TCP流量的运行状况可以认为是运行状况良好的。代码清单12-3展示了如何将TCP流量重定向到后端的MySQL。

代码清单12-3 ELB处理普通的TCP流量（不仅仅是HTTP）

```

"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    "Subnets": [{"Ref": "SubnetA"}, {"Ref": "SubnetB"}],
    "LoadBalancerName": "elb",
    "Listeners": [{
      
      "InstancePort": "3306",
      "InstanceProtocol": "TCP",
      "LoadBalancerPort": "3306",
      "Protocol": "TCP"
    }],
    "HealthCheck": {
      "Target": "TCP:3306", ) 
      "Interval": "10",
      "Timeout": "5",
      "HealthyThreshold": "3",
      "UnhealthyThreshold": "2"
    },
    "SecurityGroups": [{"Ref": "LoadBalancerSecurityGroup"}],
    "Scheme": "internal" 
  }
}

```

实际上，还可以将端口80配置为按照TCP流量进行处理，但这样你将无法根据Web服务器返回的状态代码进行运行状况检查。

## 2. 终止SSL

ELB可以用来终止SSL，而无须做任何配置。终止SSL意味着ELB提供了SSL加密的端点，将未加密的请求转发到后端服务器。图12-3展示了这是如何工作的。

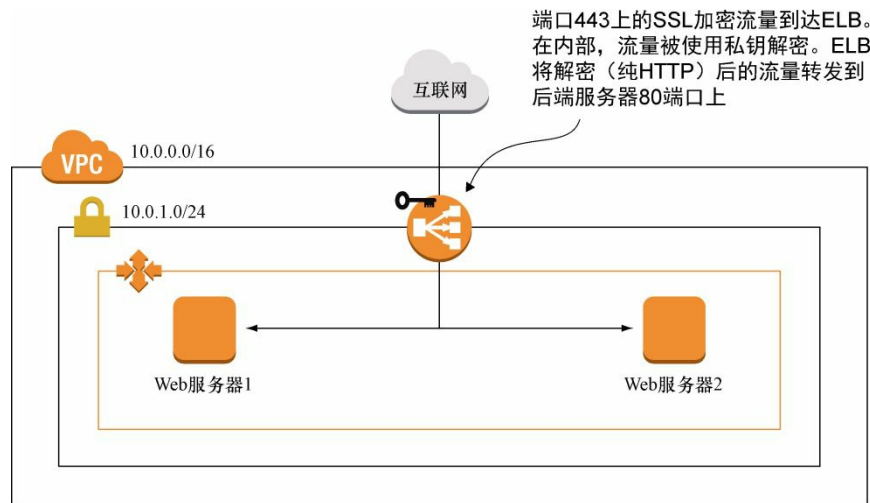


图12-3 负载均衡器可以接受加密流量，解密流量，并将未加密流量转发到后端

端口443上的SSL加密流量到达ELB。在内部，流量被使用私钥解密。ELB将解密（纯HTTP）后的流量转发到后端服务器80端口上。

客户可以使用来自AWS的预定义的安全策略以获得安全的SSL配置，好处是当SSL被发现了漏洞可以得到保护。客户可以接受来自端口443（HTTPS）的请求，而ELB终止SSL并将该请求转发到Web服务器上的端口80。这是针对SSL加密通信的简单的解决方案。SSL终止不仅适用于HTTP请求，也适用于TCP流量（如POP3、SMTP、FTP等）。

#### 注意

以下示例仅在读者已经拥有SSL证书时有效。如果没有，读者需要购买SSL证书或跳过该示例。AWS目前不提供SSL证书。读者可以使用自签名证书进行测试<sup>[1]</sup>。

<sup>[1]</sup> 2016 年发布的AWS Certificate Manager 可以帮助用户快速申请证书在AWS 资源上部署该证书。——译者注

在激活SSL加密之前，我们必须借助AWS命令行接口（Command Line Interface, CLI）将SSL证书上传到IAM：

```
$ aws iam upload-server-certificate \
--server-certificate-name my-ssl-cert \
--certificate-body file://my-certificate.pem \
--private-key file://my-private-key.pem \
--certificate-chain file://my-certificate-chain.pem
```

现在，我们可以通过引用**my-ssl-cert** 来使用SSL证书了。代码清单12-4展示了如何在ELB的帮助下配置加密的HTTP通信。

代码清单12-4 使用ELB终止SSL提供的加密通信

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    "Subnets": [{"Ref": "SubnetA"}, {"Ref": "SubnetB"}],
    "LoadBalancerName": "elb",
    "Policies": [{      <--配置SSL
      "PolicyName": "ELBSecurityPolicyName",
      "PolicyType": "SSLNegotiationPolicyType",
      "Attributes": [{
        "Name": "Reference-Security-Policy",
        "Value": "ELBSecurityPolicy-2015-05"      <--使用预定义的安全策略作为
配置
      }]
    }],
    "Listeners": [{
      "InstancePort": "80",      <--后端服务器监听端口80（HTTP）
      "InstanceProtocol": "HTTP",
      "LoadBalancerPort": "443",      <--ELB接受端口443（HTTPS）的请求
      "Protocol": "HTTPS",
      "SSLCertificateId": "my-ssl-cert",      <--引用之前上传的SSL 证书
      "PolicyNames": ["ELBSecurityPolicyName"]
    }],
    "HealthCheck": {
      [...]
    },
    "SecurityGroups": [{"Ref": "LoadBalancerSecurityGroup"}],
    "Scheme": "internet-facing"
  }
}
```

在ELB的帮助下终止SSL，将减少许多对提供安全通信至关重要的管理任务。我们鼓励你在ELB的帮助下使用HTTPS，这将保护你的客户在与你的服务器通信时免受各种攻击。

**警告**

安全策略ELBSecurityPolicy-2015-05 不再是最新的<sup>[2]</sup>。安全策略中定义了支持什么版本的SSL、支持哪些密码以及其他与安全相关的选项。如果没有使用最新的安全策略版本，则SSL设置可能会很脆弱。访问AWS官方网站可获取最新版本。

<sup>[2]</sup> 截至2017年5月，最新的安全策略是ELBSecurityPolicy-2016-08。——译者注

我们建议你仅为自己的用户提供SSL加密的通信。除保护敏感数据之外，这也将对Google的搜索排名产生积极的影响。

### 3. 记录日志

ELB可以与S3集成以提供访问日志。访问日志包含了ELB处理的所有请求的信息。你可能已经熟悉了Apache Web服务器等Web服务器的访问日志，可以使用访问日志来调试后端的故障，并分析对系统进行了多少请求。

要激活访问日志记录，ELB必须知道应将日志写入哪个S3存储桶。我们还可以指定访问日志写入S3的频率。我们需要设置一个S3存储桶的策略，以允许ELB写入存储桶，如代码清单12-5所示。

代码清单12-5 policy.json

```
{
  "Id": "Policy1429136655940",
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "Stmt1429136633762",
    "Action": ["s3:PutObject"],
    "Effect": "Allow",
    "Resource": "arn:aws:s3:::elb-logging-bucket-$YourName/ *",
    "Principal": {
      "AWS": [
        "127311923021", "027434742980", "797873946194",
        "156460612806", "054676820928", "582318560864",
        "114774131450", "783225319266", "507241528517"
      ]
    }
  ]
}
```

```
}
```

要应用策略并创建S3存储桶，要使用CLI。但不要忘记用你的姓名或昵称替换`$YourName`，以防止与其他读者发生名称上的冲突。这也适用于`policy.json`文件。

```
$ aws s3 mb s3://elb-logging-bucket-$YourName
$ aws s3api put-bucket-policy --bucket elb-logging-bucket-$YourName \
--policy file://policy.json
```

现在还可以使用代码清单12-6所示的CloudFormation描述激活访问日志。

代码清单12-6 激活ELB生成的访问日志

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    [...]
    "AccessLoggingPolicy": {
      "EmitInterval": 10,      <-- 日志写入S3 的间隔（5~60 min）
      "Enabled": true,
      "S3BucketName": "elb-logging-bucket-$YourName",      <-- S3 存储桶的名字
      "S3BucketPrefix": "my-application/production"      <-- 如果要将多个访问
      日志保存到同一个S3 存储桶（可选），可以在访问日志前加上前缀
    }
  }
}
```

ELB现在不时地将访问日志文件写入到指定的S3存储桶。访问日志类似于Apache Web服务器创建的访问日志，但是你不能更改其包含的信息的格式。以下片段显示访问日志的一行内容：

```
2015-06-23T06:40:08.771608Z elb 92.42.224.116:17006 172.31.38.190:80
0.000063 0.000815 0.000024 200 200 0 90
"GET http://elb-....us-east-1.elb.amazonaws.com:80/ HTTP/1.1"
"Mozilla/5.0 (Macintosh; ...) Gecko/20100101 Firefox/38.0" - -
```

下面是访问日志始终包含的信息片段的示例。

- 时间戳: 2015-06-23T06:40:08.771608Z 。
- ELB的名称: elb 。
- 客户端IP地址和端口: 92.42.224.116:17006 。
- 后端IP地址和端口: 172.31.38.190:80 。
- 在负载均衡器中处理请求的秒数: 0.000063 。
- 请求在后端处理的秒数: 0.000815 。
- 在负载均衡器中处理响应的秒数: 0.000024 。
- 负载均衡器返回的HTTP状态码: 200 。
- 后端返回的HTTP状态码: 200 。
- 接受的字节数: 0 。
- 发送的字节数: 90 。
- 请求: "GET http://elb-....us-east-1.elb.amazonaws.com:80/ HTTP/1.1" 。
- 用户代理: "Mozilla/5.0 (Macintosh; ...)Gecko/20100101 Firefox/38.0" 。

#### 资源清理

删除在日志示例中创建的S3存储桶:

```
$ aws s3 rb --force s3://elb-logging-bucket-$YourName
```

## 4. 跨可用区的负载均衡

ELB是一种容错服务。创建一个ELB,将收到一个公共的名称,如 **elb-1079556024.us-east-1.elb.amazonaws.com**。这个名字将作为端点(endpoint)。看到这个名字背后的东西是很有趣的,可以使用命令行应用程序**dig**(或Windows上的**nslookup**)向DNS服务器查询这个特定的名称:

```
$ dig elb-1079556024.us-east-1.elb.amazonaws.com
```

```
[...]
;; ANSWER SECTION:
elb-1079556024.us-east-1.elb.amazonaws.com. 42 IN A 52.0.40.9
elb-1079556024.us-east-1.elb.amazonaws.com. 42 IN A 52.1.152.202
[...]
```

名字**elb-1079556024.us-east-1.elb.amazonaws.com** 被解析为两个IP地址，即52.0.40.9和52.1.152.202。当创建负载均衡器时，AWS将在后台启动两个实例，并使用DNS在两者之间进行分配。为了使服务器具有容错的机制，AWS会在不同的可用区中生成负载均衡器实例。默认情况下，ELB的每个负载均衡器实例只将流量发送到同一可用区中的EC2实例。如果要跨可用性区分发请求，则可以启用跨可用区的负载均衡。图12-4展示了跨可用区负载均衡的很重要的场景。

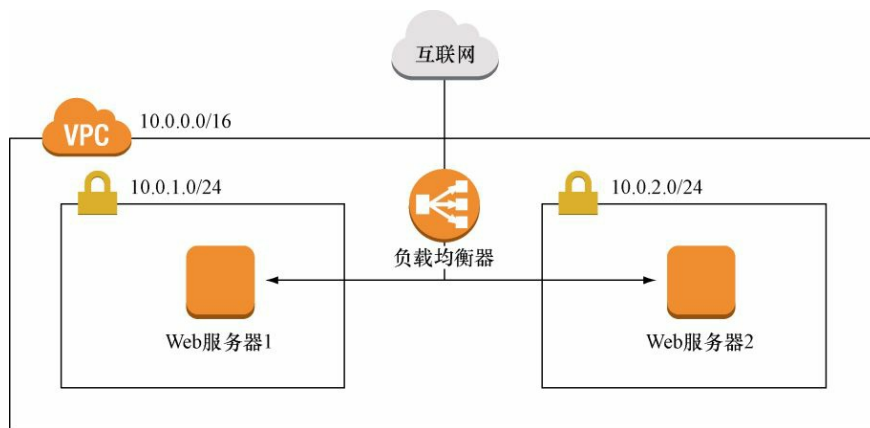


图12-4 启用跨可用区负载均衡实现在可用区之间分配流量

下面的CloudFormation代码片段展示了如何使用这个特性：

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    [...]
    "CrossZone": true
  }
}
```

我们建议启用跨可用区负载均衡（默认情况下是禁用的），以确保



请求均匀地路由到所有的后端服务器。

在下一节中，我们将了解有关异步解耦的更多信息。

## 12.2 利用消息队列实现异步解耦

利用ELB实现同步解耦是比较容易的，不需要修改代码去做到这一点。但是，对于异步解耦，必须调整代码来使用消息队列。

消息队列有一个头和一个尾。从头部读取信息时，可以向尾部添加新消息，这样可以使消息的生产和消费解耦。生产者和消费者彼此并不认识，它们只知道消息队列而已。图12-5说明了这一原则。

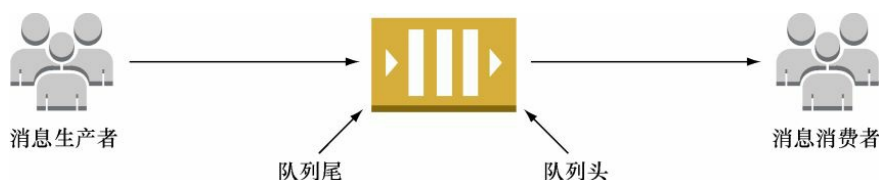


图12-5 生产者将消息发送到消息队列，消费者读取消息

我们可以将新消息放入队列，而此时并没有人正在读取消息，消息队列充当了缓冲区。为了防止消息队列增长无限大，消息只能保存一定的时间。如果我们从消息队列中消费消息，则必须确认已经成功地处理了消息，并将其从队列中永久删除。

简单队列服务（SQS）是完全托管的AWS服务。SQS提供消息队列，确保消息被传递至少一次。

- 极少数情况下，一条消息可能被消费两次。如果将这一点与其他消息队列产品进行比较，你可能会觉得有点儿奇怪。但在本章的后面我们看到了如何处理这一问题。
- SQS并不保证消息的顺序，因此你可能会按照不同的顺序读取消息。

SQS的这个限制也是有其好处的：

- 你可以随意添加多个消息到SQS中。
- 消息队列随着生产和消费的消息数量而伸缩。

这个服务的定价模式也很简单：每个SQS请求需要客户支付0.000 000 50美元，即每百万个请求需要支付 0.5美元。生产消息是一个请

求，消费则是另一个请求（如果消息的有效负载大于64 KB，则每64 KB存储块就算作一个请求）。

## 12.2.1 将同步过程转换成异步过程

典型的同步过程如下：用户向你的服务器发出请求，服务器完成一些处理，并将结果返回给用户。为了使这个过程看起来更具体，我们将在下面的例子中讨论创建URL预览图片的过程。

- （1）用户提交URL。
- （2）服务器下载URL中的内容，并将其转换为PNG格式的图片。
- （3）服务器将PNG文件返回给用户。

利用一个小技巧，可以将这个过程转化成异步的。

- （1）用户提交URL。
- （2）服务器将包含随机ID和URL的消息发入消息队列中。
- （3）服务器返回一个将来可以访问这个PNG图片的链接，该链接包含随机ID（`http:// $Bucket.s3-website-us-east-1.amazonaws.com/$RandomId.png`）。
- （4）在后端，工作进程从队列中读取消息、下载内容，并将内容转化为PNG格式的图片，然后将图片上传到S3。
- （5）在某个时间点，用户尝试通过已知的链接下载这个PNG文件。

如果要使用异步的处理过程，就必须管理过程启动程序跟踪过程状态的方式。一种方法是将ID返回给可用于查找过程的启动器。在此过程中，ID一步一步传递。

## 12.2.2 URL2PNG应用的架构

我们现在可以建立一个简单但是解耦的软件片段，我们称其为URL2PNG。它的功能是将一个网页的URL转换成PNG图片。接下来，我们将使用Node.js来完成编程的部分，并且将会用到SQS。图12-6展示了URL2PNG应用是如何工作的。

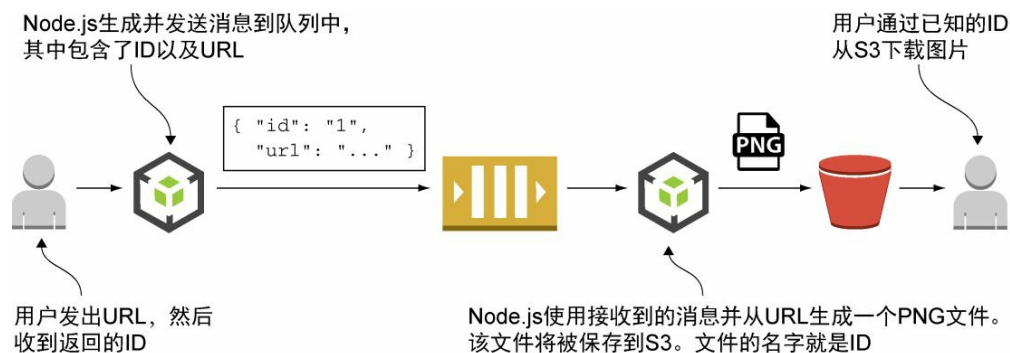


图12-6 URL2PNG是如何工作的

为了完成这个例子，我们需要创建一个启用Web托管的S3存储桶。执行下面的命令，用我们的名字或者昵称替换`$YourName`，以避免与其他读者的名字冲突。

```
$ aws s3 mb s3://url2png-$YourName
$ aws s3 website s3://url2png-$YourName --index-document index.html \
--error-document error.html
```

Web托管的特性是需要的，以便用户可以从S3下载图片。现在可以开始建立消息队列了。

## 12.2.3 创建消息队列

创建消息队列是一件简单的事情——仅需要指定队列的名字：

```
$ aws sqs create-queue --queue-name url2png
{
  "QueueUrl": "https://queue.amazonaws.com/878533158213/url2png"
}
```

返回的QueueUrl 将会在后面的例子中用到，一定要保存好。

## 12.2.4 以程序化的方法处理消息

你现在已经有了一个可以用来发送消息的SQS队列了。为了生成消息，你需要指定队列以及有效的载荷（payload）。你将再次使Node.js与AWS SDK结合使用，将你的程序与AWS连接起来。

### 安装并开始使用Node.js

要安装Node.js请访问Node.js官方网站，并下载满足所用的操作系统的包。

下面是通过适用于Node.js的AWS SDK生成消息的方式。它将在以后被URL2PNG的工作进程所使用。Node.js脚本可以像下面这样用（现在不要尝试运行这个命令——需要先安装和配置URL2PNG）：

```
$ node index.js "http://aws.amazon.com"
PNG will be available soon at
http://url2png-$YourName.s3-website-us-east-1.amazonaws.com/XYZ.png
```

像之前的一样，读者可以在下载的源代码中找到这段代码。URL2PNG示例位于/chapter12/url2png/下面。代码清单12-7展示了index.js的实现。

代码清单12-7 index.js: 发送消息到队列中

```
var AWS = require('aws-sdk');
var uuid = require('node-uuid');
var sqs = new AWS.SQS({      <--- 创建一个SQS 端点
    "region": "us-east-1"
});
if (process.argv.length !== 3) {      <--- 检查是否提供了URL
    console.log('URL missing');
```

```

    process.exit(1);
}

var id = uuid.v4();      <-- 创建一个随机ID
var body = {            <-- 消息的内容中包含随机ID 和URL
    "id": id,
    "url": process.argv[2]
};

var params = {
    "MessageBody": JSON.stringify(body),      <-- 创建一个SQS 端点
    "QueueUrl": "$QueueUrl"                 <-- 将消息内容转换为JSON 字符串
};

sqs.sendMessage(params, function(err) {      <-- 发送消息的队列（创建队列时返回的）
    if (err) {
        console.log('error', err);
    } else {
        console.log('PNG will be available soon at http://url2png-$YourName.s3
-
        ➡ website-us-east-1.amazonaws.com/' + id + '.png');
    }
});

```

在运行这段脚本之前，需要先安装Node.js模块。在终端中运行**npm install**来完成依赖模块的安装。我们将会发现一个名为**config.json**的文件，这个文件需要修改。确保将**QueueUrl**改为你在本示例开头创建的队列，并将**Bucket**改为**url2png-\$YourName**。

现在就可以使用**nodeindex.js "http://aws.amazon.com"**来运行脚本了。程序会作出类似于“PNG will be available soon at **http://url2png-\$YourName.s3-website-us-east-1.amazonaws.com/XYZ.png**”这样的响应。要验证消息是否已准备好被使用，可以查询队列中有多少条消息：

```

$ aws sqs get-queue-attributes \
--queue-url $QueueUrl \
--attribute-names ApproximateNumberOfMessages
{
  "Attributes": {
    "ApproximateNumberOfMessages": "1"
  }
}

```

```
}  
}
```

接下来，是时候处理消费消息的工作进程了，这个处理要完成生成PNG文件的所有工作。

## 12.2.5 程序化地消费消息

使用SQS处理消息需要3个步骤。

- (1) 接收消息。
- (2) 处理消息。
- (3) 确认消息被成功处理。

现在，我们就来实现上述步骤来将URL变成PNG。

要从SQS队列接收消息，必须指定以下内容：

- 队列。
- 要接受的最大的消息的数量（为了获得更高的吞吐量，可以批量处理消息）。
- 要从队列中取出这条消息来处理所花的秒数（在这期间，还必须从队列中删除该消息，否则该消息将再次被接收）。
- 希望等待接受消息的最大秒数（从SQS接收消息是通过轮询API来实现的，但是API允许的轮询最长时间是10 s）。

代码清单12-8展示了如何使用SDK去实现从队列接收一条消息。

代码清单12-8 worker.js: 从队列中接收一条消息

```
var fs = require('fs');  
var AWS = require('aws-sdk');  
var webshot = require('webshot');  
var sqs = new AWS.SQS({
```

```

    "region": "us-east-1"
  });
  var s3 = new AWS.S3({
    "region": "us-east-1"
  });

  function receive(cb) {
    var params = {
      "QueueUrl": "$QueueUrl",
      "MaxNumberOfMessages": 1,      <---一次消费不超过一条消息
      "VisibilityTimeout": 120,     <---在120 s 的时间内从队列中获取消息
      "WaitTimeSeconds": 10         <---轮询10 s 等待新的消息
    };
    sqs.receiveMessage(params, function(err, data) {      <---对SQS 调用receive
Message 操作
      if (err) {
        cb(err);
      } else {
        if (data.Messages === undefined) {      <---检查是否有可用的消息
          cb(null, null);
        } else {
          cb(null, data.Messages[0]);          <---获取一条且是唯一的一条消息
        }
      }
    });
  }
}

```

接收步骤现在已经实现，下一步就应该是处理消息了，如代码清单12-9所示。得益于一个名为**webshot** 的Node.js模块，创建一个网站的屏幕截图就成为一件很容易的事。

代码清单12-9 worker.js: 处理消息（得到截屏图并上传到S3上）

```

function process(message, cb) {
  var body = JSON.parse(message.Body);      <---消息正文是一个JSON 字符串，将
其转换为JavaScript 对象
  var file = body.id + '.png';
  webshot(body.url, file, function(err) {    <---使用webshot 模块创建截屏图
    if (err) {
      cb(err);
    } else {
      fs.readFile(file, function(err, buf) {  <---打开由webshot 模块保存到
本地磁盘的截屏图
        if (err) {

```





```

function run() {
  receive(function(err, message) {      <---接收一条消息
    if (err) {
      throw err;
    } else {
      if (message === null) {          <---调用deleteMessage 操作
        console.log('nothing to do');
        setTimeout(run, 1000);        <---检查消息是否可用
      } else {
        console.log('process');

        process(message, function(err) {      <---在1 s 内再次调用run 方法
          if (err) {
            throw err;
          } else {
            acknowledge(message, function(err) {      <---处理消息确认消息
              if (err) {
                throw err;
              } else {
                console.log('done');
                setTimeout(run, 1000);      <---在1 s 内再次调用run 方法
              }
            });
          }
        });
      }
    }
  });
}

run();      <---调用run 方法启动程序

```

现在，我们就可以启动工作进程来处理已在队列中的消息了。使用**node worker.js**来运行脚本，应该看到输出的一些内容，说明工作进程正处在处理步骤，然后就会转换到完成状态。几秒钟之后，截屏图应该被上传到S3。我们的第一个异步应用程序已经完成。

我们创建了一个异步解耦的应用程序。如果URL2PNG服务流行起来，并且数以百万计的用户开始使用它，那么队列就会变得越来越长。因为你的工作进程无法从URL生成许多个PNG。很酷的事情是，我们可以增加尽可能多的工作进程来处理这些消息。不是只启动1个工作进

程，而是启动10个或者100个。另一个优点是，如果工作进程因某种原因终止，那么在2 min后，正在“飞行”中的消息将可用于消费，并由另一个工作进程接管。这就具有了容错的特性！如果将系统设计为异步解耦，则系统易于扩展，而且具有良好的容错基础。第13章将集中讨论这个话题。

#### 资源清理

按照以下的步骤删除消息队列：

```
$ aws sqs delete-queue --queue-url $QueueUrl
```

同时，不要忘记清理并删除示例中使用的S3存储桶。发出以下命令，用你的名字替换\$YourName：

```
$ aws s3 rb --force s3://url2png-$YourName
```

## 12.2.6 SQS消息传递的局限性

本章前面提到了SQS的一些局限性。本节将对此进行更详细的介绍。

### 1. SQS不保证消息仅被传送一次

如果在**VisibilityTimeout** 期间未能删除收到的消息，则该消息将被再次接收。这个问题可以通过使接收幂等来解决。幂等意味着无论消息的消费频率如何，结果都保持不变<sup>[3]</sup>。在URL2PNG示例中，设计如下：如果多次处理消息，则将相同的图片上传到S3。如果图片在S3上已经可用，则会被替换。幂等有效解决了分布式系统中的许多问题，保证了消息至少传递一次。

<sup>[3]</sup> 在编程中，幂等操作的特点是其任意多次执行所产生的影响均与一次执行产生的影响一样。——译者注

不是所有的东西都可以做成幂等的，发送电子邮件就是一个很好的例子。如果你多次处理邮件，并且每次都会发送一封电子邮件，那么收件人就会为此而烦恼。作为解决的方法，你可以使用数据库跟踪自己是否已经发送了这封电子邮件。

在很多情况下，至少有一次是很好的权衡的结果。在使用SQS之前检查你的要求，确认这种权衡的处理符合你的需求。

## 2. SQS并不保证消息的顺序

消费消息可能与生成消息的顺序不同。如果需要严格的顺序，就要寻找其他方法了。SQS是容错和可扩展的消息队列。但如果需要的是稳定的消息顺序，那么将很难找到像SQS一样具有扩展能力的解决方案。我们的建议是改变系统的设计，使你不再需要稳定的顺序或在客户端生成顺序。

## 3. SQS不会取代消息代理

SQS不是一个类似ActiveMQ的消息代理——SQS只是一个消息队列服务。不要指望SQS具有消息代理提供的功能。将SQS与ActiveMQ进行对比就好像将DynamoDB与MySQL进行对比一样<sup>[4]</sup>。

<sup>[4]</sup> 2017 年AWS 发布了基于Apache ActiveMQ 的名为AmazonMQ 的新服务。——译者注

## 12.3 小结

- 解耦使事情变得更容易了，因为它减少了依赖。
- 同步解耦需要双方同时使用，但双方彼此并不了解对方。
- 通过异步解耦，可以在不要求双方可用的情况下进行通信。
- 大多数应用程序可以使用ELB提供的负载均衡服务，在不改变代码的情况下实现同步解耦。
- 负载均衡器可以定期对你的应用进行运行状况检查，以确定后端是否准备好处理流量。
- 异步解耦仅在异步处理过程中可用。但是在大多数情况下，可以将同步处理修改为异步处理。
- 使用SQS实现异步解耦要求使用SDK进行编程。

## 第13章 容错设计

### 本章主要内容

- 什么是容错，为什么需要容错
- 使用冗余消除单点故障
- 失败后重试
- 使用幂等操作实现失败后重试
- AWS服务保证

磁盘、网络、电源等出现故障是不可避免的。容错可以解决这个问题。容错系统就是为故障而构建的。如果发生故障，容错系统将不会中断，并且可以继续处理请求。如果系统有单点故障，它就不是容错的。用户可以通过在系统中引入冗余来实现容错，并通过将各系统分离，使一方不依赖另一方，从而正常运行。

要使系统容错，最便捷的方式就是组成容错模块系统。如果所有的模块是容错，那么系统就是容错的。许多AWS服务默认情况下就是容错，尽可能使用这些服务。要不然需要自己处理故障。

遗憾的是，AWS其中一个非常重要的服务，即EC2，默认情况下并不是容错的。虚拟机不是容错的。这意味着，在默认情况下，单台EC2服务不是容错的。但是AWS提供了组件来解决这个问题。这个解决方案包括了自动扩展组（auto-scaling group）、ELB和SQS。

区别各种服务，保证下列要求是非常重要的。

- 没有（单点故障）——在出现故障时，不能处理请求。
- 高可用性——在出现故障时，需要一些时间直到像之前一样处理请求。
- 容错——在出现故障时，请求会像之前一样得到处理，并且没有任何可用性問題。

以下是本书中涵盖的AWS服务的详细保证。单点故障（SPOF）意

味着，如一个硬件发生故障，那么服务则中断。

- **Amazon EC2实例** ——单个EC2可能由于各种原因而失败，如硬件故障、网络问题、可用区问题等。然而使用自动扩展组使得一组EC2以冗余的方式处理请求，以实现高可用性或容错。
- **Amazon关系数据库服务（RDS）单个实例** ——单个RDS实例失败的原因有很多，如硬件故障、网络问题、可用性区域问题等。然而使用多可用区模式部署能实现高可用性。

高可用性（HA）意味着当故障发生时，服务在短时间内不可用，但会自动回到正常状态。

- **弹性网络接口（Elastic Network Interface, ENI）** ——网络接口绑定到可用区（AZ），因此，如果可用区不可用，那么网络接口将不可用。
- **Amazon虚拟私有云（Virtual Private Cloud, VPC）子网** ——VPC子网绑定到一个AZ，因此，如果此AZ不可用，那么子网将关闭。在不同的AZ中可以使用多个子网来消除对单个AZ的依赖。
- **Amazon弹性块状存储（Elastic Block Store, EBS）卷** ——EBS卷绑定到AZ，如果AZ不可用，数据卷将不可用（你的数据不会丢失）。你可以定期创建EBS快照，以便可以在另一个AZ中重新创建EBS卷。
- **Amazon关系数据库服务（Relational Database Service, RDS）多可用区实例** ——在多可用区模式下运行时，如果主实例发生故障，会更改DNS记录切换到备用实例，需要短暂的停机时间（1 min）。

容错意味着如果发生故障，将不会影响到：

- **弹性负载均衡（Elastic Load Balancing, ELB）**，需要部署到至少两个可用区
- **Amazon EC2安全组**；
- **带有ACL和路由器表的Amazon虚拟私有云（Virtual Private Cloud, VPC）**；
- **弹性IP地址（Elastic IP Address, EIP）**；
- **Amazon 简单存储服务（Simple Storage Service, S3）**；
- **Amazon Elastic Block Store（EBS）快照**；
- **Amazon DynamoDB**；

- Amazon CloudWatch;
- 自动扩展组;
- Amazon Simple Queue Service (SQS);
- AWS Elastic Beanstalk;
- AWS OpsWorks;
- AWS CloudFormation;
- AWS身份和访问管理 (IAM, 不绑定到单个区域; 如果创建IAM用户, 该用户在所有区域都可用)。

为什么要关心容错? 因为最终容错系统为终端用户提供高质量的服务。无论系统中发生什么, 用户都不会受到影响, 可以继续消费内容, 购买物品或与朋友交谈。几年前, 实现容错是昂贵的, 但在AWS中, 提供容错系统是一个可负担的标准。

#### 本章要求

要充分理解本章, 你需要阅读并理解以下概念:

- EC2 (第3章);
- 自动扩展 (第11章);
- 弹性负载均衡 (第12章);
- SQS (第12章)。

该示例大量使用以下内容:

- Elastic Beanstalk (第5章);
- DynamoDB (第10章);
- Express, 一个Node.js Web应用程序框架。

在本章中, 我们将了解基于EC2实例设计容错Web应用程序所需的所有知识 (默认情况下不是容错的)。



## 13.1 使用冗余EC2实例提高可用性

值得一提的是，EC2实例本身不是容错的。虚拟机下是主机系统（host system）。你的虚拟机可能遭受由主机系统导致的崩溃，主要来自以下几个原因。

- 如果主机硬件出现故障，则它无法托管在其之上的虚拟机。
- 如果去往（或者来自）主机的网络连接中断，虚拟机也将失去网络通信的能力。
- 如果主机系统与电源断开连接，则虚拟机也会关闭。

但是在虚拟机上运行的软件也可能导致崩溃。

- 如果软件存在内存泄漏，内存会被耗尽，可能是一天、一个月、一年或更长时间，但最终它会耗尽。
- 如果软件将数据写入到磁盘但从不删除，将最终耗尽磁盘空间。
- 应用程序可能无法正常边界情况，也会崩溃。

导致崩溃的原因，无论是主机还是软件，单个EC2实例都是单点故障。如果你依赖单个EC2实例，你的系统有可能会崩溃——唯一的问题是什么时候会发生而已。

### 13.1.1 冗余可以去除单点故障

设想制造蓬松云馅饼的生产线。生产蓬松的云馅饼需要几个生产步骤（步骤已简化）。

- （1）制作馅饼皮。
- （2）冷却馅饼皮。
- （3）把蓬松的云馅饼的调料均匀地铺洒在馅饼皮上。
- （4）冷却蓬松的云馅饼。

### （5）包装蓬松的云馅饼。

当前的设置是单个生产线。这个设置有一个大问题：当其中一个步骤崩溃时，整个生产线必须停止。图13-1说明了当第二步（冷却馅饼皮）崩溃时的问题。以下步骤不再工作，因为他们不会收到冷却的馅饼皮。

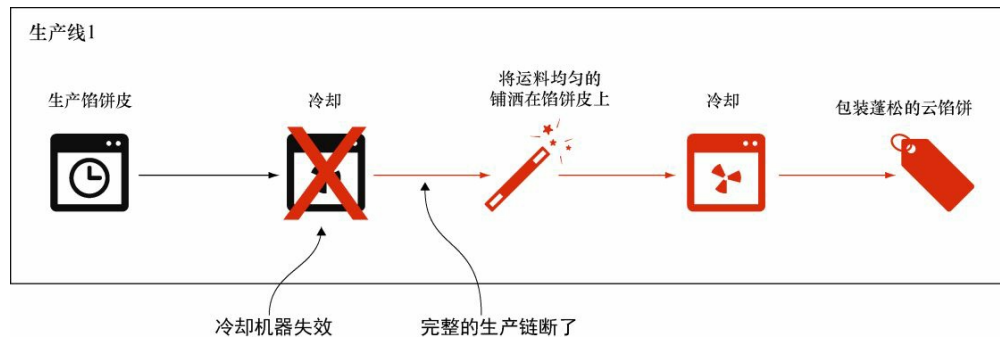


图13-1 单点故障影响的不仅仅是自己本身，而是整个系统

为什么不能有多条生产线呢？而不是用一个，假设我们有3个。如果其中一条线路出现故障，另外两条线路仍然可以为世界上所有饥饿的客户生产蓬松的云馅饼。图13-2展示了这一改进，唯一的缺点是，我们需要3倍的机器。

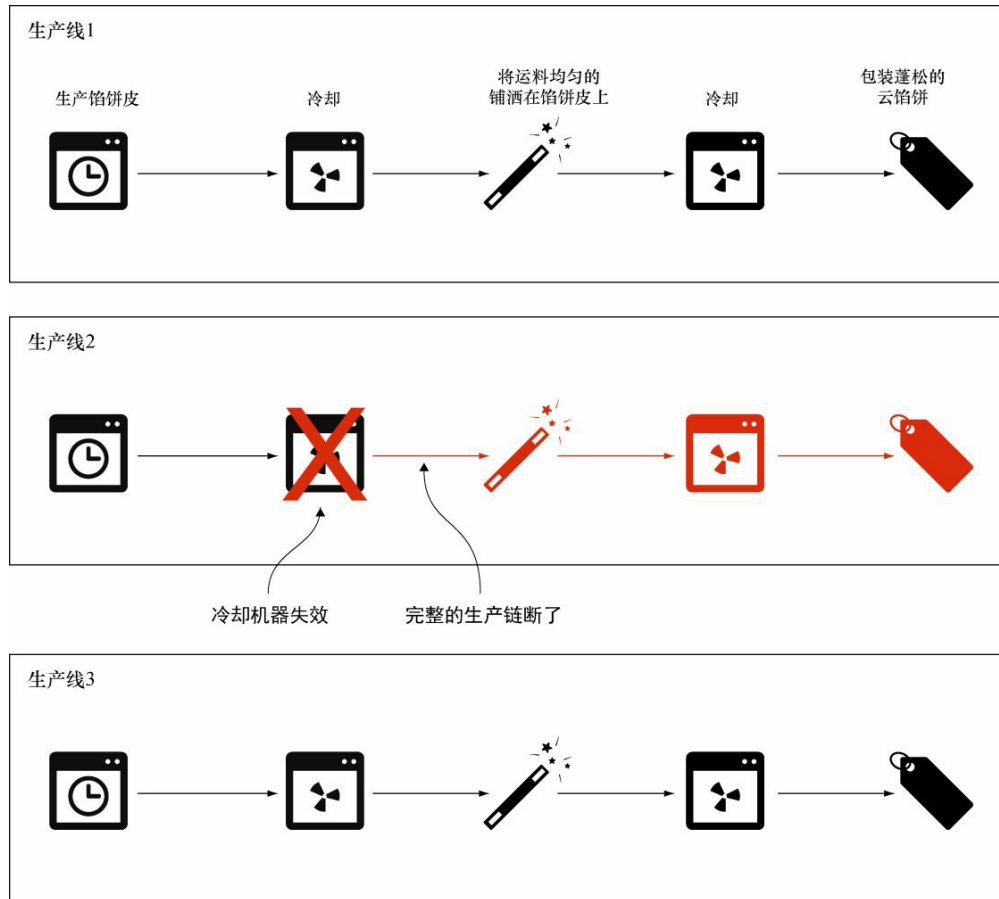


图13-2 冗余消除单点故障使系统更加健壮

该例子也可以用来解释EC2实例。可以有3个实例运行软件，而不是只有一个EC2实例。如果其中一个实例崩溃，其他两个实例仍然能够处理传入的请求。你还可以最小化一个实例对3个实例造成的成本影响：可以选择3个小实例，而不是一个大型EC2实例。动态服务器池出现的问题是，如何与实例通信？答案是解耦：在EC2实例和请求者之间放置负载均衡器或者消息队列。接下去将了解这是如何工作的。

### 13.1.2 冗余需要解耦

图13-3展示了如何通过冗余和同步解耦来使得EC2达到容错。如果其中一个EC2实例崩溃，ELB将停止向失败的实例发送请求。自动扩展组在几分钟内替换失败的EC2实例，并且ELB开始将请求路由到新的实例上。

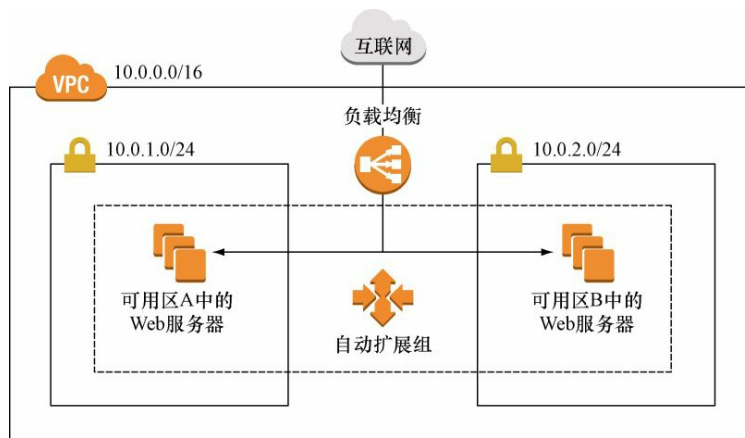


图13-3 具有自动扩展组和ELB的容错EC2服务器

再看一下图13-3，看看哪些部分是多余的。

- 可用区 ——使用两个。如果一个AZ崩溃，仍然有EC2实例在其他AZ运行。
- 子网 ——子网隶属于AZ。因此，在每个AZ中需要一个子网，子网也是冗余的。
- EC2实例 ——EC2实例多冗余。在单个子网（AZ）中有多个实例，在两个子网（两个AZ）中有实例。

图13-4展示了使用EC2构建的容错系统，该系统使用冗余和异步解耦的功能来处理来自SQS队列的消息。

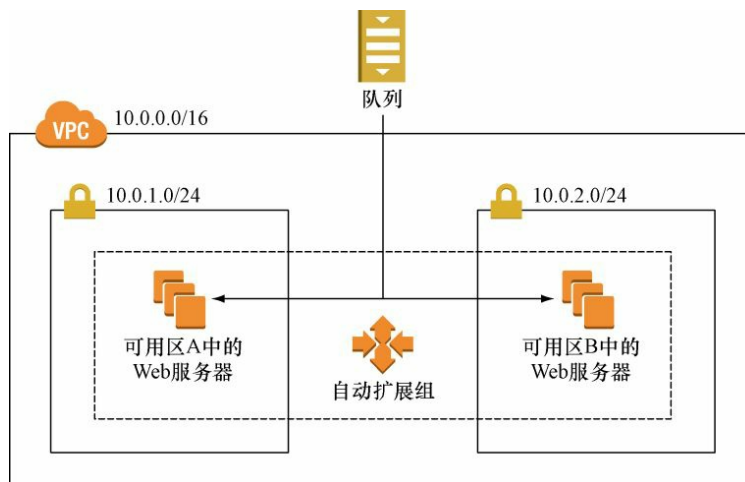


图13-4 具有自动扩展组和SQS的容错的EC2服务器

在这两张图中，负载均衡/SQS队列只出现了一个。这并不意味着

ELB或SQS是单点故障;相反，ELB和SQS默认是容错的。

## 13.2 使代码容错的注意事项

如果想要容错，必须在代码中实现它。我们可以通过遵循本节中提出的两个建议，将错误容错设计到代码中。

### 13.2.1 让其崩溃，但也重试

Erlang编程语言以“让其崩溃”（let it crash）这个概念而闻名。这只是意味着每当程序不知道该怎么办，它就崩溃，有人需要处理崩溃。最常见的是，人们忽视了一个事实，即Erlang也是以重试（retry）出名。崩溃而不重试是无用的——如果你无法从崩溃的情况下恢复，你的系统将关闭，这不是想要的。

我们可以将“让其崩溃”的概念（有些人称之为“快速故障”）应用到同步解耦和异步解耦场景。在同步解耦场景中，请求的发送者必须实现重试逻辑。如果在一定时间内没有返回响应，或返回错误，则发送者通过再次发送相同的请求来重试。在异步解耦场景中，事情更容易。如果消息被消耗，但在一定时间内未被确认，则它返回到队列。下一个消费者抓住消息并再次处理它。默认情况下，重试被内置到异步系统中。

“让其崩溃”在所有情况下并不是都没有用。如果程序要响应告诉发送方该请求包含无效内容，这不是让服务器崩溃的原因：结果将保持不变，无论你重试多少次。但是，如果服务器无法到达数据库，重试很有意义。在几秒钟内，数据库可能再次可用，并能够成功处理重试的请求。

重试不是那么容易。想象一下，你想重试一个博客文章的创建。每次重试都将创建数据库中的一个新条目，其中包含与以前相同的数据。在数据库中最终有很多重复。防止这将涉及一个重要的概念，接下来将介绍：幂等重试。

### 13.2.2 幂等重试使得容错成为可能

如何防止博客文章由于重试而被多次添加到数据库？一个简单的方法是使用标题作为主键。如果主键已经使用，你可以假定该帖子已经在数据库中，并跳过将其插入数据库的步骤。现在插入博客帖子是幂等的，这意味着无论多么频繁地应用某个动作，结果必须是相同的。在当前示例中，结果是数据库条目。

让我们尝试一个更复杂的例子。插入博客帖子在现实中更复杂，过程看起来像下面这样。

- (1) 在数据库中创建博客帖子条目。
- (2) 无效缓存，因为数据已更改。
- (3) 发布博客的Twitter feed的链接。

让我们仔细看一下每一步。

## 1. 在数据库中创建一个博客条目

之前我们已经涵盖了这一步骤，使用主题作为主键（primary key）。但是这一次，我们使用全局唯一标识符（UUID），而不是用主题作为主键。一个UUID就像**550e8400-e29b-11d4-a716-446655440000**是一个随机的ID，由客户端生成。由于UUID本身的一些特性，不会生成两个一样的UUID。如果想创建一个博客条目，必须将包含UUID、主题和文本的请求发给ELB。ELB将请求路由到后端其中一个实例上。后端的实例检查主键是否存在。如果不存在，数据库中将增加一条新的记录。如果存在，插入就继续。图13-5展示了这个流程创建一个博客条目对于幂等操作是一个很好的例子。

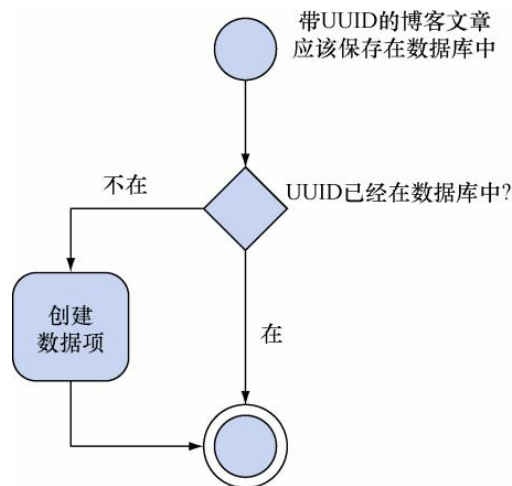


图13-5 幂等的数据库插入操作：在数据库中创建博客文章

可以使用数据库来解决这个问题。只需发送一个插入到你的数据库。有以下3件事可能会发生。

- 数据库插入数据。步骤成功完成。
- 数据库响应错误，主键已在使用。步骤成功完成。
- 数据库以不同的错误响应。该步骤崩溃。

仔细想想实现幂等的最佳方式！

## 2. 让缓存失效

此步骤向缓存层发送无效消息。你不需要担心太多幂等性：如果缓存比所需的更频繁，则不会受到影响。如果缓存无效，则下一次请求命中缓存时，缓存不包含数据，将查询源站（在这里指数据库）返回结果。然后将结果放入缓存中以便后续请求。如果由于重试使缓存多次失效，最糟糕的事情是需要再次调用数据库。非常简单。

## 3. 发送到博客的Twitter feed

要使此步骤幂等，你需要使用一些技巧，因为你与不支持幂等操作与第三方交互。遗憾的是，没有解决方案将保证你只发布一个状态更新到Twitter。你可以保证创建是至少有一个（一个或多个）状态更新，或至多一个（一个或无）状态更新。一个简单的方法是向Twitter API请求



最新的状态更新，如果其中一个匹配你要发布的状态更新，则跳过该步骤，因为它已经完成。

但是，Twitter是一个最终一致性的系统：不能保证你发布后立即看到状态更新。你可以最终发布多次状态更新。另一种方法是在数据库中保存是否已经发布了博客帖子状态更新。但想象一下，保存到你发布到Twitter的数据库，然后向Twitter API发出请求，但刚好那一刻，系统崩溃。数据库提示说Twitter的状态更新已发布，但实际情况却不是。你需要做出选择：允许丢失状态更新，或者允许多个状态更新。提示：这是一个商业决策。图13-6展示了两种解决方案的流程。

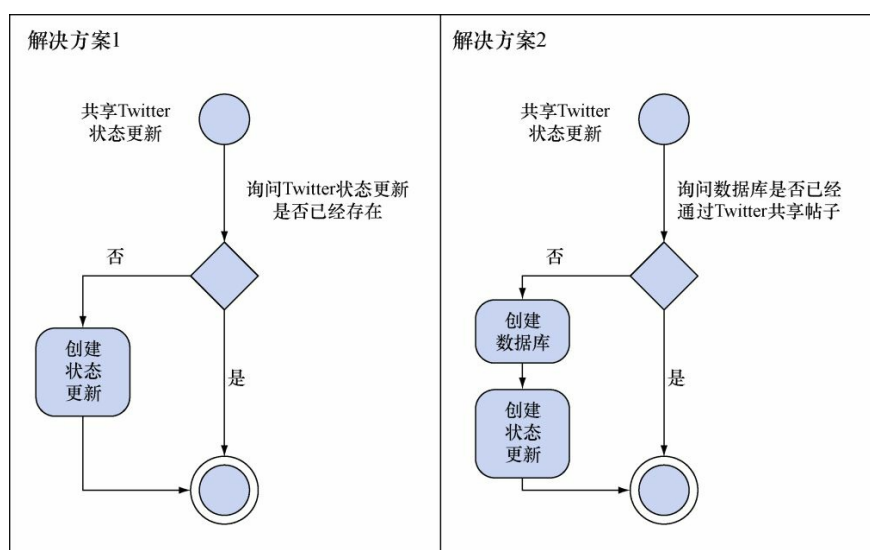


图13-6 幂等的Twitter状态更新：如果它还没有完成，只共享状态更新

举个例子，我们将在AWS上设计、实施和部署分布式容错Web应用程序。这个例子将综合本书中的大部分知识，演示分布式系统的工作方式。

## 13.3 构建容错Web应用：Imagery

在开始架构和设计容错的Imagery Web应用之前，我们将简要介绍应用程序最后应该做什么。用户应该能够上传图片，然后这个图片用深褐色过滤器转换，使其看起来变旧。用户可以查看深褐色图片。图13-7展示了这一过程。

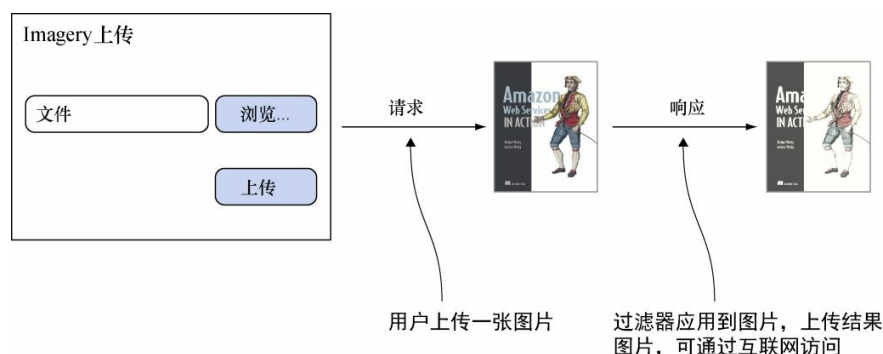


图13-7 用户上传图片到Imagery，其中应用了过滤器

图13-7展示了这一过程的问题——它是同步的。如果服务器在请求和响应期间死机，则不会处理用户的图片。在许多用户想要使用Imagery应用程序时会出现另一个问题：系统变得很繁忙，可能会变慢或停止工作。因此，这个过程应该变成异步的。第12章通过使用SQS消息队列介绍了异步解耦的思想，如图13-8所示。

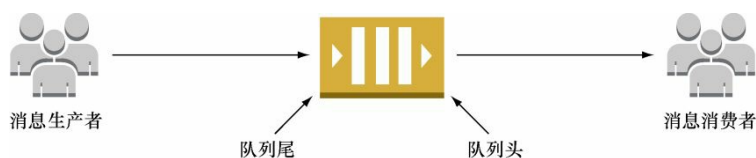


图13-8 生产者发送消息到消息队列，消费者读取消息

当设计异步过程时，过程的跟踪非常重要。我们需要一些类型的标识符。当用户想要上传图片时，用户首先创建进程。此进程创建会返回唯一的ID。使用ID，用户能够上传图片。如果图片上传完成，服务器开始在后台处理图片。用户可以随时查找进程。处理图片时，用户看不到深褐色图片。但是一旦图片被处理，查找进程返回深褐色图片。图13-9展示了异步过程。

现在有一个异步过程，是时候把组件映射到AWS服务上去了。请记住，AWS上的大多数服务默认情况下都是容错的，因此尽可能使用它们。图13-10展示了一种方法。

为了使事情尽可能简单，所有操作都可以通过REST API来访问，REST API将由EC2实例提供。最后，EC2实例将提供进程并调用所有AWS服务，如图13-10所示。

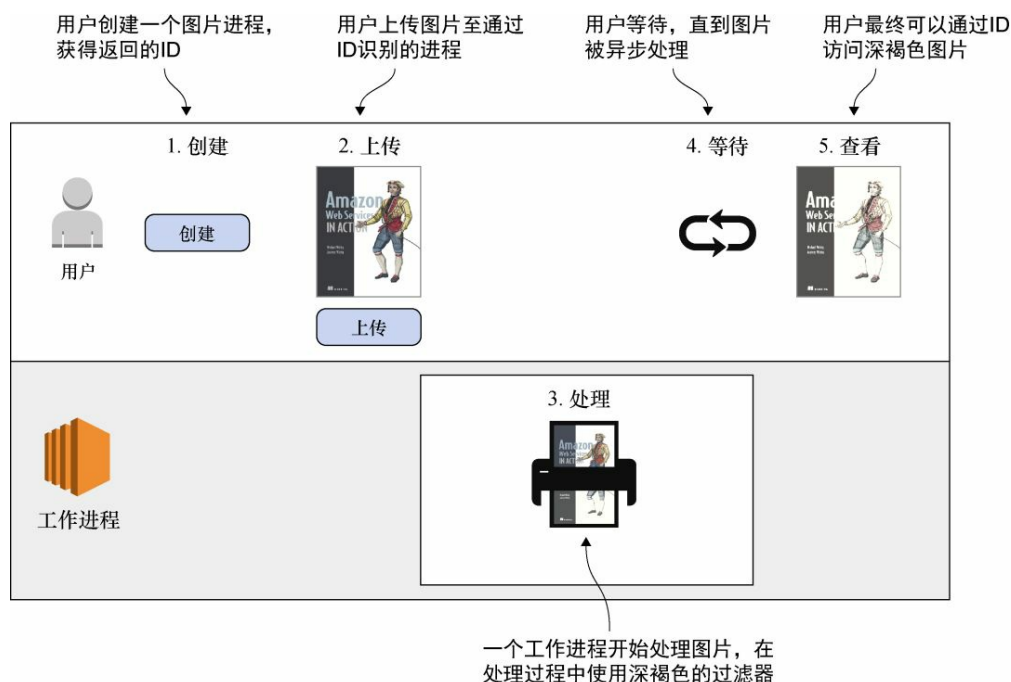


图13-9 用户异步上传图片至Imagery，其中应用了过滤器

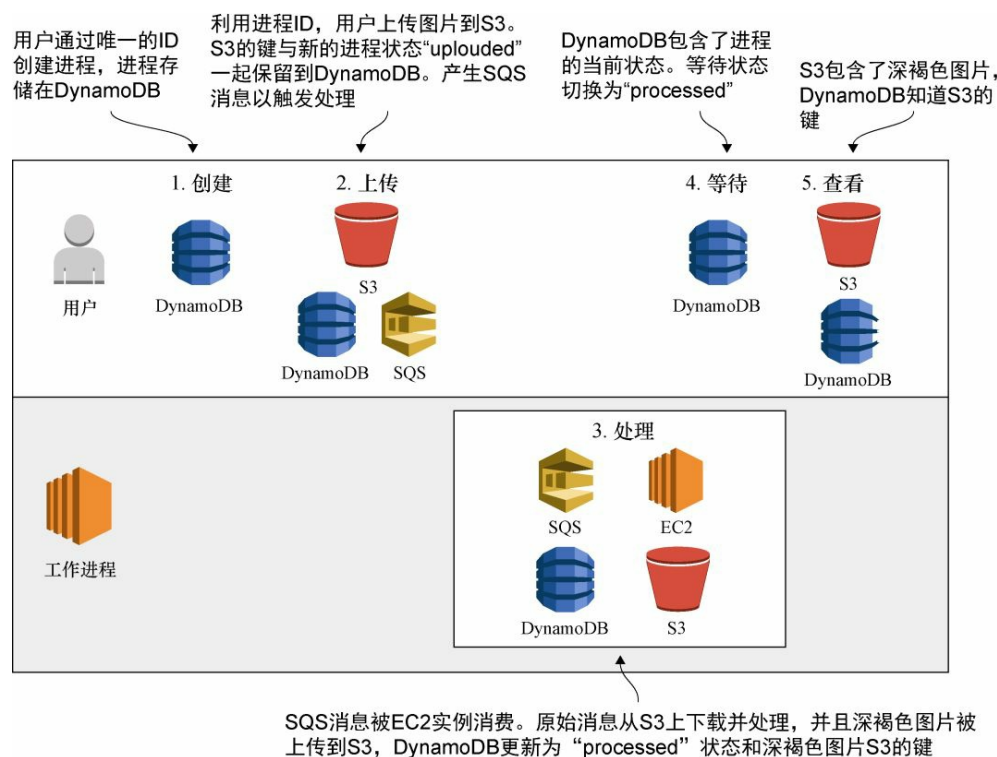


图13-10 组合AWS服务实现异步Imagery过程

我们将使用许多AWS服务来实现Imagery应用程序。它们中的大多数在默认情况下是容错的，但EC2不是。你将使用幂等的图片状态机（image-state machine）来处理这个问题，图片状态机会在下一节介绍。

#### 示例都包含在免费套餐中

本章中的所有示例都包含在免费套餐中。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

#### AWS Lambda和Amazon API Gateway即将到来

AWS正在开发名为Lambda的服务。使用Lambda，你可以将代码函数上传到AWS，然后在AWS上执行该函数。你不需要再维护EC2实例，只需要关心代码。AWS Lambda用于短时间运行进程（最多60 s），因此不能使用Lambda创建Web服务器。但是AWS将提供许多集成钩子（hook）。例如，每当一个对象被添加到S3时，AWS就可以触发一个Lambda函数，或者当新消息到达SQS时触发Lambda函数。遗憾的是，AWS Lambda在编写本书时并不适用于AWS所有地区，因此我们决定不包括此服务的介绍。

Amazon API网关使你能够运行REST API，而无须运行任何EC2实例。你可以指定每当收到GET/ some/resource 请求时，它将触发一个Lambda函数。Lambda和Amazon API Gateway的组合可以构建强大的服务，无须维护单个EC2实例。遗憾的是，在本书编写时，Amazon API Gateway并不适用于所有地区。

## 13.3.1 幂等图片状态机

幂等的图片状态机（image-state machine）听起来很复杂。我们需要花一些时间来解释它，因为它是Imagery应用程序的核心。让我们来看一下什么是状态机，在这个上下文中什么是幂等。

### 1. 有限状态机

状态机具有至少一个开始状态和一个结束状态（这里讨论的是有限状态机）。在开始状态和结束状态之间，状态机可以具有许多其他状态。机器还定义状态之间的转换。例如，具有3个状态的状态机可能如下所示：

(A) -> (B) -> (C).

这意味着：

- 状态A是开始状态；
- 过渡从状态A到B；
- 过渡从状态B到C；
- 状态C是结束状态。

但是在(A) → (C)或(B) → (A)之间没有可能的转换，Imagery状态机看起来像下面这样：

(Created)-> (Uploaded)-> (Processed)

创建了新进程（状态机）之后，唯一的转换可能就是**Uploaded**。要进行此转换，需要上传的原始图片的S3键。**Created** 到**Uploaded** 的转换可以通过**uploaded(s3Key)** 定义。基本上，这个过渡同样适用于**Uploaded** 到**Processed** 的转换。这个转换可以用深褐色图片的S3键来完成：**processed(s3Key)**。

不要混淆，因为上传和图片过滤处理不会出现在状态机中。这些是发生的基本动作，但我们只对结果感兴趣，我们不跟踪行动的进展。该过程不清楚10%的数据已上传或30%的图片完成处理。它只关心操作是否100%完成。我们可以想象一堆可以实现的其他状态，但这个例子中略过，只介绍调整大小（**Resized**）和共享（**Shared**）两个例子。

## 2. 幂等状态转换

无论转换发生的频率如何，幂等状态转换都必须具有相同的结果。如果知道状态转换是幂等的，可以使用这样一个简单的技巧：在转换过程中失败的情况下，重试整个状态转换。

接下去看看需要实现的两个状态转换。第一个转换**Created** 到**Uploaded** 可以像下面这样实现（伪代码）：

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== "Created") {
    throw new Error("transition not allowed")
  }
  DynamoDB.updateItem(processId, {"state": "Uploaded", "rawS3Key": s3Key})
  SQS.sendMessage({"processId": processId, "action": "process"});
}
```

这个实现的问题是它不是幂等的。想象一下，**SQS.sendMessage** 失败了。如果重试，状态转换将失败。但第二次调用**Uploaded(s3Key)** 将抛出一个“transition not allowed”错误，因为**DynamoDB.updateItem** 在第一次调用期间成功。

为了解决这个问题，需要更改**if** 语句使函数幂等：

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== "Created" && process.state !== "Uploaded") {
    throw new Error("transition not allowed")
  }
  DynamoDB.updateItem(processId, {"state": "Uploaded", "rawS3Key": s3Key})
  SQS.sendMessage({"processId": processId, "action": "process"});
}
```

如果现在重试，将对DynamoDB进行多次更新，这不会受到影响。并且可以发送多个SQS消息，这也不会受到影响，因为SQS消息消费者也必须是幂等的。这同样适用于转换“Uploaded”为“Processed”。

接下来，我们将开始实施Imagery服务器。

## 13.3.2 实现容错Web服务

将Imagery应用程序分为两部分：服务器和工作程序。服务器负责向用户提供REST API，工作程序负责处理消耗的SQS消息和处理映像。

### 代码在哪里下载

像之前一样，在下载源代码中可以找到相关代码，图片位于chapter13中。

服务器将支持以下路由。

- **POST /image** ——当执行这个路由，一个新的图片进程被创建。
- **GET /image/:id** ——返回进程状态，由路径参数: id 指定。
- **POST /image/:id/upload** ——此路由为使用path 参数指定的: id 进程提供文件上传。

要实现这一服务，会再次用到Node.js和Express Web应用程序框架。在这个项目中只会使用到Express框架，因此不会觉得麻烦。

### 1. 设置服务器项目

和往常一样，需要一些样例代码来加载依赖，初始AWS端点以及类似的东西，如代码清单13-1所示。

代码清单13-1 初始化Imagery服务（server/server.js）

```
var express = require('express');    <--加载Node.js 模块（依赖）
var bodyParser = require('body-parser');
var AWS = require('aws-sdk');
var uuid = require('node-uuid');
var multiparty = require('multiparty');

var db = new AWS.DynamoDB({          <--创建DynamoDB 端点
  "region": "us-east-1"
});
var sqs = new AWS.SQS({              <--创建SQS 端点
  "region": "us-east-1"
});
var s3 = new AWS.S3({                <--创建S3 端点
  "region": "us-east-1"
});
var app = express();                 <--创建Express 应用

app.use(bodyParser.json());           <--告诉Express 解析请求主体

[...]
```

```
app.listen(process.env.PORT || 8080, function() {    <--在环境变量PORT 上
  // 启动Express，默认端口是8080
  console.log("Server started. Open http://localhost:" +
    (process.env.PORT || 8080) + " with browser.");
});
```

不用太过于担心这个样例代码，有趣的部分后面很快就会介绍。

## 2. 创建一个新的Imagery进程

为了提供REST API来创建图片进程，一组EC2实例在负载均衡器后运行Node.js代码。图片进程存储在DynamoDB中。图13-11展示了创建新的图片的请求流程。



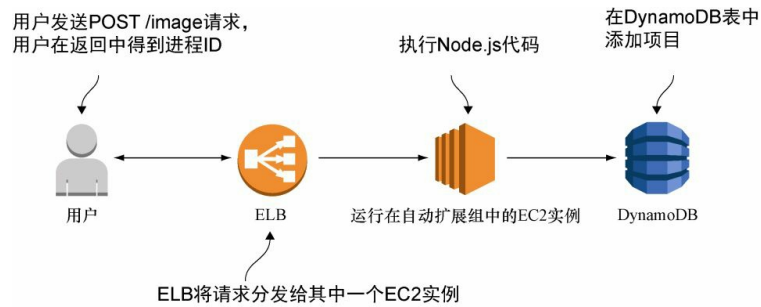


图13-11 在Imagery中创建一个新图片的流程

我们现在在Express应用中增加路由来处理POST /image 请求，如代码清单13-2所示。

代码清单13-2 Imagery服务：POST /image创建图片流程

```

app.post('/image', function(request, response) {    <---在Express 中注册路由
  var id = uuid.v4();    <---为进程创建一个唯一ID
  db.putItem({    <---对DynamoDB 调用putItem 操作
    "Item": {
      "id": {
        "S": id    <---id 属性是DynamoDB 的主键
      },
      "version": {
        "N": "0"    <---使用乐观锁定的版本（乐观锁定在下面会有解释）
      },
      "state": {
        "S": "created"    <---进程现在处于创建状态，当状态更换发生时这一属性会变化
      }
    },
    "TableName": "imagery-image",    <---DynamoDB 表在本章的后面会创建
    "ConditionExpression": "attribute_not_exists(id)"
  }, function(err, data) {    <---如果项已经存在就阻止替换
    throw err;
  } else {
    response.json({"id": id, "state": "created"});    <---以进程ID作为响应
  }
});
});

```

现在可以创建一个新进程了。

#### 乐观锁定

要防止对DynamoDB项目进行多次更新，可以使用名为乐观锁定（**optimistic locking**）的技巧。当你要更新项目时，必须告知要更新的版本。如果该版本与数据库中项目的当前版本不匹配，则更新将被拒绝。

想象以下场景。在版本0中创建一个项目。进程A查找该项目（版本0）。进程B也查找该项目（版本0）。现在，进程A想通过在DynamoDB上调用**updateItem** 操作来进行更改。因此，进程A指定预期版本为0。DynamoDB因为版本匹配将允许修改，但DynamoDB也会将项目的版本更改为1，因为执行了更新。现在，进程B想进行修改，并向DynamoDB发送请求，期望项目版本为0。DynamoDB将拒绝该修改，因为预期版本与DynamoDB知道的版本不匹配，即1。

要解决进程B的问题，可以使用前面介绍的同样的技巧——重试。进程B将再次查找该项目，现在在版本1中，并且可以（你希望）进行更改。

乐观锁定有一个问题：如果许多修改并行发生，则会产生很多开销，因为重试的次数很多。但是这只是一个问题，如果你期望大量并发写入单个项目，这可以通过更改数据模型解决。Imagery应用程序中不是这样。只有少数写入预期发生在一个单一的项目：乐观锁定是一个完美的适合，以确保你没有两个写入，其中一个操作会覆盖另一个更改。

乐观锁定的相反是悲观锁定。可以通过使用信号量来实现悲观锁策略。在更改数据之前，需要锁定信号量。如果信号量已经被锁定，则等待信号量再次变空。

我们需要实现的下一个路由是查找进程的当前状态。

### 3. 查找Imagery进程

我们现在增加路由到Express应用中来处理GET `/image/:id` 请求。图13-12展示了这一请求流程。

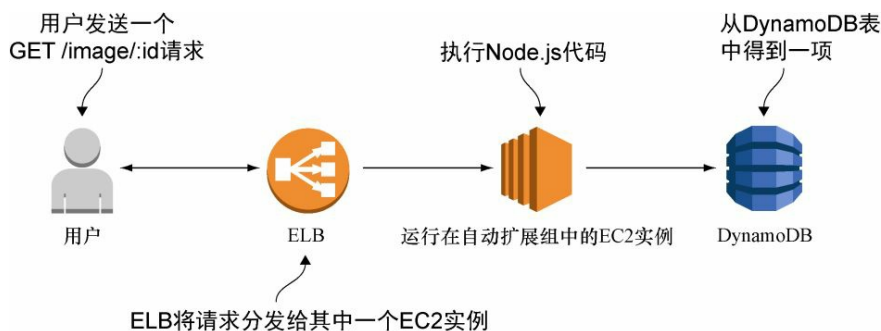


图13-12 在Imagery中查看图片进程返回状态

Express将通过在`request.params.id` 中提供它来处理路径参数:`id`。这一实现需要基于路径参数ID从DynamoDB获取项目，如代码清单13-3所示。

代码清单13-3 Imagery服务器: GET `/image/:id` 查找图片进程

```
function mapImage(item) {      <--辅助函数将DynamoDB 结果映射到JavaScript 对象
  return {
    "id": item.id.S,
    "version": parseInt(item.version.N, 10),
    "state": item.state.S,
    "rawS3Key": [...],
    "processedS3Key": [...],
    "processedImage": [...]
  };
};

function getImage(id, cb) {
  db.getItem({      <--对DynamoDB 调用getItem 操作
    "Key": {
      "id": {      <--id 是主键散列值
        "S": id
      }
    },
    "TableName": "imagery-image"
  }, function(err, data) {
    if (err) {
      cb(err);
    } else {
      if (data.Item) {
        cb(null, mapImage(data.Item));
      } else {
        cb(new Error("image not found"));
      }
    }
  });
}

app.get('/image/:id', function(request, response) {      <--使用Express 注册路由
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      response.json(image);      <--以image 进程来响应
    }
  })
}
```

```
});  
});
```

唯一缺少的是上传部分，接下来看一下上传。

## 4. 上传图片

通过POST 请求上传图片需要执行以下几个步骤。

- (1) 将原始图片上传到S3。
- (2) 修改DynamoDB的项目。
- (3) 发送SQS消息触发图片处理。

图13-13展示了这一 workflow。

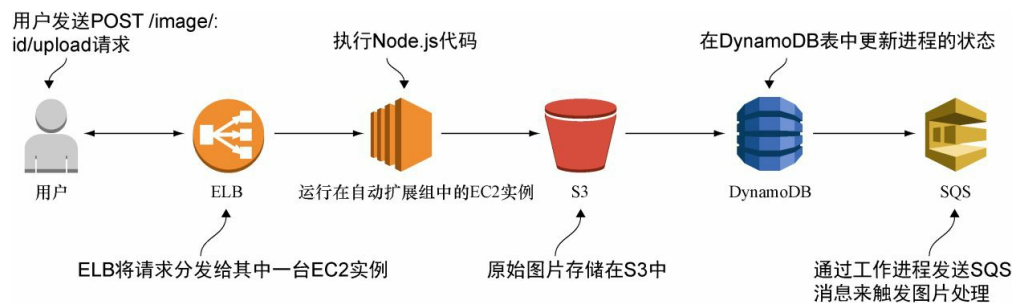


图13-13 上传原始图片到Imagery来触发图片处理

代码清单13-4展示了这些步骤的具体实现。

代码清单13-4 Imagery服务器: POST /image/:id/upload 上传图片

```
function uploadImage(image, part, response) {  
    var rawS3Key = 'upload/' + image.id + '-' + Date.now();    <---为S3 对象  
    创建一个键  
    s3.putObject({    <---对S3 调用putObject  
        "Bucket": process.env.ImageBucket,    <---S3 桶名作为环境变量传入（该桶将  
        在本章后面部分中创建）  
        "Key": rawS3Key,
```

```

    "Body": part,      <--Body 部分是上传的数据流
    "ContentLength": part.byteCount
  }, function(err, data) {
    if (err) {
      throw err;
    } else {
      db.updateItem({      <--对Dynamodb 调用updateItem
        "Key": {
          "id": {
            "S": image.id
          }
        },
        "UpdateExpression": "SET #s=:newState,      <--更新状态、版本和原始S3
        密钥
        ➡ version=:newVersion, rawS3Key=:rawS3Key",
        "ConditionExpression": "attribute_exists(id)      <--仅当项目存在、版本
        本等于预期版本且状态是允许的状态之一时更新
        ➡ AND version=:oldVersion
        ➡ AND #s IN (:stateCreated, :stateUploaded)",
        "ExpressionAttributeNames": {
          "#s": "state"
        },
        "ExpressionAttributeValues": {
          ":newState": {
            "S": "uploaded"
          },
          ":oldVersion": {
            "N": image.version.toString()
          },
          ":newVersion": {
            "N": (image.version + 1).toString()
          },
          ":rawS3Key": {
            "S": rawS3Key
          },
          ":stateCreated": {
            "S": "created"
          },
          ":stateUploaded": {
            "S": "uploaded"
          }
        },
        "ReturnValues": "ALL_NEW",
        "TableName": "imagery-image"
      }, function(err, data) {
        if (err) {
          throw err;
        } else {

```

```

    sqs.sendMessage({      <--对SQS 调用sendMessage
      "MessageBody": JSON.stringify({
        "imageId": image.id,
        "desiredState": "processed"      <--消息包含进程ID
      }),
      "QueueUrl": process.env.ImageQueue,      <--队列URL 通过环境变量传
递
    }, function(err) {
      if (err) {
        throw err;
      } else {
        response.json(lib.mapImage(data.Attributes));
      }
    });
  });
}
});
}
});
}

app.post('/image/:id/upload', function(request, response) {      <--使用Express注册路由
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      var form = new multiparty.Form();      <--“魔法代码”处理图片上传
      form.on('part', function(part) {
        uploadImage(image, part, response);
      });
      form.parse(request);
    }
  });
});
});

```

服务器端完成。接下来，将继续在Imagery工作进程中实现处理部分，然后就可以部署应用程序了。

### 13.3.3 实现容错的工作进程来消费SQS消息

Imagery工作进程在后台执行异步的工作：在应用过滤器的同时将

图片处理为深褐色。工作程序处理消耗的SQS消息和处理图片。幸运的是，消耗SQS消息是Elastic Beanstalk解决的常见任务，稍后将使用它来部署应用程序。Elastic Beanstalk可以配置为侦听SQS消息并对每个消息执行HTTP POST 请求。最后，工作程序实现了由Elastic Beanstalk调用的REST API。要实现工作进程，你将再次使用Node.js和Express框架。

## 1. 设置服务器项目

如代码清单13-5所示，与以往一样，需要样例代码来加载依赖关系、初始化AWS端点等。

代码清单13-5 初始化Imagery工作进程（worker/worker.js）

```
var express = require('express');      <--加载Node.js 模块（依赖）
var bodyParser = require('body-parser');
var AWS = require('aws-sdk');
var assert = require('assert-plus');
var Caman = require('caman').Caman;
var fs = require('fs');

var db = new AWS.DynamoDB({           <--创建DynamoDB 端点
  "region": "us-east-1"
});
var s3 = new AWS.S3({                 <--创建S3 端点
  "region": "us-east-1"
});

var app = express();                  <--创建Express 应用
app.use(bodyParser.json());

app.get('/', function(request, response) {    <--注册返回空对象的健康状态检查的路由
  response.json({});
});

[...]

app.listen(process.env.PORT || 8080, function() {    <--在由环境变量PORT
定义的端口或者默认端口8080 上启动Express
  console.log("Worker started on port " + (process.env.PORT || 8080));
});
```

Node.js模块caman用于创建深褐色图片，接下来我们会用到这个。

## 2. 处理SQS消息和处理图片

SQS消息触发原始图片处理，这由工作进程控制。一旦接收到消息，工作进程开始从S3下载原始图片，应用深褐色过滤器，并将处理的图片上传回S3。之后，DynamoDB中的处理状态将被修改。图13-14展示了这些步骤。

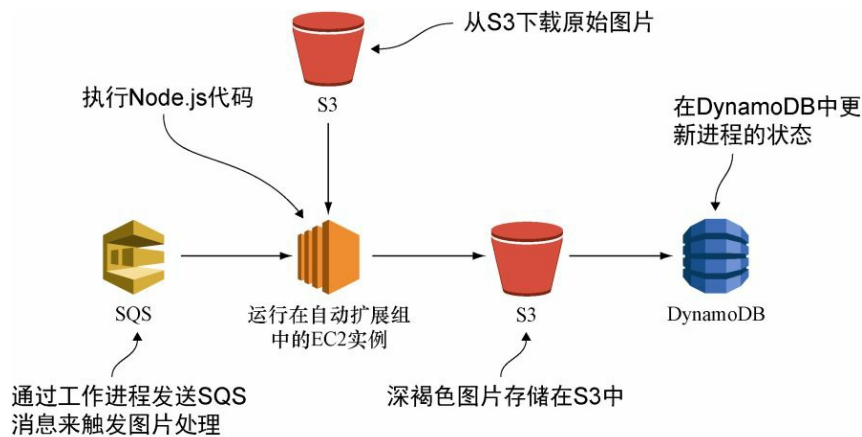


图13-14 处理原始图片，将深褐色图片上传到S3

如果不直接从SQS接收消息，可以采取一个捷径。Elastic Beanstalk是将要使用的部署工具，它提供了消费队列中消息的功能，并为每个消息调用HTTP POST 请求。配置对资源/sqs 进行的POST 请求。代码清单13-6展示了具体实现。

代码清单13-6 Imagery工作进程：POST /sqs 处理SQS消息

```
function processImage(image, cb) {    <---processImag 的实现在这里没有展示，
可以在本书的源文件夹中找到
    var processedS3Key = 'processed/' + image.id + '-' + Date.now() + '.png'
    ;
    // download raw image from S3
    // process image
    // upload sepia image to S3
    cb(null, processedS3Key);
}

function processed(image, request, response) {
```



```

processImage(image, function(err, processedS3Key) {
  if (err) {
    throw err;
  } else {
    db.updateItem({      ←--对DynamoDB 调用updateItem 操作
      "Key": {
        "id": {
          "S": image.id
        }
      },
      "UpdateExpression": "SET #s=:newState,      ←--更新状态、版本和已处理
的S3 键
      ➡ version=:newVersion, processedS3Key=:processedS3Key",
      "ConditionExpression": "attribute_exists(id)      ←--仅当项目存在、版本
本等于预期版本且状态是允许的状态之一时更新
      ➡ AND version=:oldVersion
      ➡ AND #s IN (:stateUploaded, :stateProcessed)",
      "ExpressionAttributeNames": {
        "#s": "state"
      },
      "ExpressionAttributeValues": {
        ":newState": {
          "S": "processed"
        },
        ":oldVersion": {
          "N": image.version.toString()
        },
        ":newVersion": {
          "N": (image.version + 1).toString()
        },
        ":processedS3Key": {
          "S": processedS3Key
        },
        ":stateUploaded": {
          "S": "uploaded"
        },
        ":stateProcessed": {
          "S": "processed"
        }
      },
      "ReturnValues": "ALL_NEW",
      "TableName": "imagery-image"
    }, function(err, data) {
      if (err) {
        throw err;
      } else {
        response.json(lib.mapImage(data.Attributes));      ←--以进程的新状

```

```

    态作为响应
    }
    });
  }
});
}

app.post('/sqs', function(request, response) {    <---使用Express 注册路由
  assert.string(request.body.imageId, "imageId");
  assert.string(request.body.desiredState, "desiredState");
  getImage(request.body.imageId, function(err, image) {    <---getImage 的
实现与服务器上的相同
    if (err) {
      throw err;
    } else {
      if (request.body.desiredState === 'processed') {
        processed(image, request, response);    <---如果SQS 消息的disired St
ate是“processed”，调用processed函数
      } else {
        throw new Error("unsupported desiredState");
      }
    }
  });
});
});

```

如果POST/sqs 返回2XX的HTTP状态码，Elastic Beanstalk会认为消息传递成功，并从队列中删除消息，否则消息重新提交。

现在可以处理SQS消息来处理原始图片，并将深褐色图片上传到S3。接下去是以容错方式将所有代码部署到AWS。

### 13.3.4 部署应用

如之前所述的，将使用Elastic Beanstalk部署服务器和工作进程。将使用CloudFormation模板。这可能听起来貌似很奇怪，因为用自动化工具来使用另一个自动化工具。但CloudFormation能做的比部署两个Elastic Beanstalk应用程序还有点多。它定义如下：

- 用于存储原始图片和处理图片的S3桶；

- DynamoDB表imagery-image；
- SQS队列和死信队列（dead-letter queue）；
- 用于服务器和EC2工作进程实例的IAM角色；
- 用于服务器和工作进程的Elastic Beanstalk应用。

创建CloudFormation堆栈需要相当长的时间，这就是为什么应该这么做。创建堆栈后，查看模板。之后，堆栈就可以使用了。

为了部署Imagery，我们创建了CloudFormation模板（位于<https://s3.amazonaws.com/awsinaction/chapter13/template.json>）。基于该模板创建一个堆栈，该堆栈输出EndpointURL 返回一个可以从浏览器访问使用Imagery的URL地址。这是如何从终端创建堆栈。

```
$ aws cloudformation create-stack --stack-name imagery \
--template-url https://s3.amazonaws.com/\
awsinaction/chapter13/template.json \
--capabilities CAPABILITY_IAM
```

现在我们来查看一下CloudFormation模板。

## 1. 部署S3、DynamoDB和SQS

代码清单13-7所示的CloudFormation片段描述了S3桶、DynamoDB表和SQS队列。

代码清单13-7 Imagery CloudFormation模板：S3、DynamoDB和SQS

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "AWS in Action: chapter 13",
  "Parameters": {
    "KeyName": {
      "Description": "Key Pair name",
      "Type": "AWS::EC2::KeyPair::KeyName",
      "Default": "mykey"
    }
  },
  "Resources": {
    "Bucket": {      <-- S3 存储桶用于上传和处理图片，启用Web 托管
```

```

    "Type": "AWS::S3::Bucket",
    "Properties": {
      "BucketName": {"Fn::Join": ["-",
        ["imagery", {"Ref": "AWS::AccountId"}]}],    <---桶名包含账号ID确保
    唯一性
      "WebsiteConfiguration": {
        "ErrorDocument": "error.html",
        "IndexDocument": "index.html"
      }
    },
    "Table": {    <---包含图片进程的DynamoDB 表
      "Type": "AWS::DynamoDB::Table",
      "Properties": {
        "AttributeDefinitions": [{
          "AttributeName": "id",
          "AttributeType": "S"
        }],
        "KeySchema": [{
          "AttributeName": "id",    <---id 属性用于主键散列值
          "KeyType": "HASH"
        }],
        "ProvisionedThroughput": {
          "ReadCapacityUnits": 1,
          "WriteCapacityUnits": 1
        },
        "TableName": "imagery-image"
      }
    },
    "SQSDLQueue": {    <---SQS 队列用于接收无法处理的消息
      "Type": "AWS::SQS::Queue",
      "Properties": {
        "QueueName": "message-dlq"
      }
    },
    "SQSQueue": {    <---SQS 队列触发图片处理
      "Type": "AWS::SQS::Queue",
      "Properties": {
        "QueueName": "message",
        "RedrivePolicy": {    <---如果一条消息接收超过10 次，移至死信队列
          "deadLetterTargetArn": {"Fn::GetAtt":
            ["SQSDLQueue", "Arn"]},
          "maxReceiveCount": 10
        }
      }
    },
    [...]

```

```

    },
    "Outputs": {
      "EndpointURL": {      <--在浏览器中用输出地址访问Imagery
        "Value": {"Fn::GetAtt": ["EBServerEnvironment", "EndpointURL"]},
        "Description": "Load Balancer URL"
      }
    }
  }
}

```

死信队列（dead-letter queue, DLQ）的概念在这里也需要简要介绍一下。如果无法处理单个SQS消息，则该消息在其他工作进程的队列中再次变为可见，这称为重试。但是如果由于某种原因重试失败（也许代码中有错误），消息将永远驻留在队列中，并可能因为许多重试浪费大量的资源。为了避免这种情况，可以配置死信队列。如果消息被重试超过特定次数，将从原始队列中删除并转移到DLQ。其区别是DLQ上没有消息的工作线程。但是，如果DLQ包含多个消息，则应创建一个CloudWatch警报，因为需要通过查看DLQ中的消息手动查看此问题。

现在已经设计了基本资源，接下去转到更加具体的资源。

## 2. 用于服务器和工作中的EC2实例的IAM角色

请记住，仅授予所需的权限很重要。所有服务器必须能够执行以下操作。

- 模板中创建`sqs:SendMessage` 发送到SQS队列触发图片处理。
- 模板中创建`s3:PutObject` 上传文件到S3（可以进一步限制上传key的前缀）。
- 模板中在DynamoDB表中创建`dynamodb:GetItem`、`dynamodb:PutItem` 和`dynamodb:UpdateItem`。
- `cloudwatch:PutMetricData`，这是Elastic Beanstalk需要的。
- `s3:Get`、`s3:List` 和`s3:PutObject` 这些是Elastic Beanstalk需要的。

所有工作进程实例能够处理以下事项。

- 模板中在SQS队列中创建`sqs:ChangeMessageVisibility`

- `sqs:DeleteMessage` 和 `sqs:ReceiveMessage` 。
- 模板中创建 `s3:PutObject` 上传文件到S3（可以进一步限制上传键的前缀）。
- 模板中在DynamoDB表中创建 `dynamodb:GetItem` 和 `dynamodb:UpdateItem` 。
- `cloudwatch:PutMetricData`，这是Elastic Beanstalk需要的。
- `s3:Get`、`s3:List` 和 `s3:PutObject`，这是Elastic Beanstalk需要的。

如果对IAM角色感到不满意，可以在下载的源代码中查看本书的代码。模板的IAM角色可以在chapter13/template.json中找到。

接下来开始设计Elastic Beanstalk应用。

### 3. 用于服务器的**Elastic Beanstalk**

简短回顾Elastic Beanstalk，在5.3节中曾经讲过。Elastic Beanstalk由以下元素组成。

- 应用程序 是逻辑容器。它包含版本，环境和配置。要在区域中使用AWS Elastic Beanstalk，必须首先创建应用程序。
- 版本 包含应用程序的特定版本。要创建新版本，必须将可执行文件（打包到归档文件中）上传到S3。一个版本基本上是一个指向这个可执行文件的指针。
- 配置模板 包含默认配置。可以使用自定义配置模板管理应用程序的配置（如应用程序侦听的端口）以及环境配置（如虚拟机的大小）。
- 环境 是AWS Elastic Beanstalk执行应用程序的地方。环境包括一个版本和配置。为一个应用程序运行多个环境可以多次使用版本和配置。

图13-15展示了Elastic Beanstalk应用程序的各个部分。

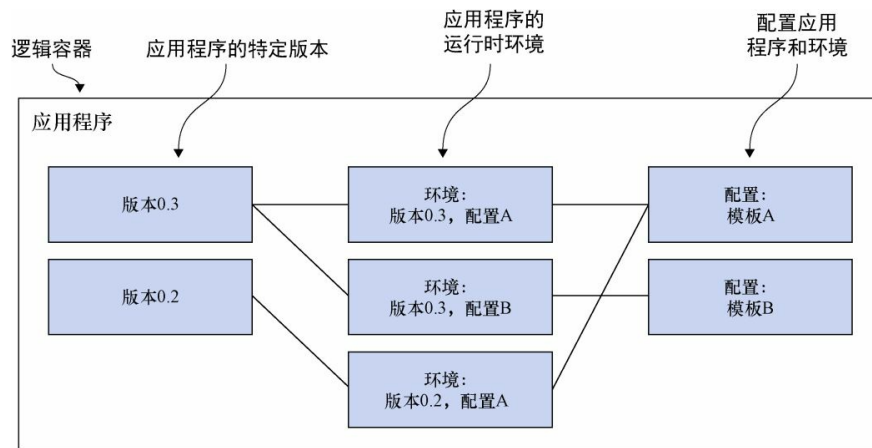


图13-15 AWS Elastic Beanstalk应用程序包含版本、配置和环境

现在已经刷新内存，我们来看一下Imagery服务器的Elastic Beanstalk应用，如代码清单13-8所示。

代码清单13-8 Imagery CloudFormation模板：用于服务器的Elastic Beanstalk

```
"EBServerApplication": {      <--描述服务器应用程序容器
  "Type": "AWS::ElasticBeanstalk::Application",
  "Properties": {
    "ApplicationName": "imagery-server",
    "Description": "Imagery server: AWS in Action: chapter 13"
  }
},
"EBServerConfigurationTemplate": {
  "Type": "AWS::ElasticBeanstalk::ConfigurationTemplate",
  "Properties": {
    "ApplicationName": {"Ref": "EBServerApplication"},
    "Description": "Imagery server: AWS in Action: chapter 13",
    "SolutionStackName":
      "64bit Amazon Linux 2015.03 v1.4.6 running Node.js",      <--使用AmazonLinux 2015.03版本运行Node.js 0.12.6
    "OptionSettings": [{
      "Namespace": "aws:autoscaling:asg",
      "OptionName": "MinSize",
      "Value": "2"      <--为了容错，最少两个EC2 实例
    }, {
      "Namespace": "aws:autoscaling:launchconfiguration",
      "OptionName": "EC2KeyName",
      "Value": {"Ref": "KeyName"}      <--传递键值对中的参数的值
    }, {
      "Namespace": "aws:autoscaling:launchconfiguration",
      "OptionName": "IamInstanceProfile",
```

```

    "Value": {"Ref": "ServerInstanceProfile"}    <---连接到在前一节中创建的
IAM 实例的配置文件
  }, {
    "Namespace": "aws:elasticbeanstalk:container:nodejs",
    "OptionName": "NodeCommand",
    "Value": "node server.js"    <---启动命令
  }, {
    "Namespace": "aws:elasticbeanstalk:application:environment",
    "OptionName": "ImageQueue",
    "Value": {"Ref": "SQSQueue"}    <---将SQS 队列传递到环境变量
  }, {
    "Namespace": "aws:elasticbeanstalk:application:environment",
    "OptionName": "ImageBucket",
    "Value": {"Ref": "Bucket"}    <---将S3 存储桶传递到环境变量
  }, {
    "Namespace": "aws:elasticbeanstalk:container:nodejs:staticfiles",
    "OptionName": "/public",
    "Value": "/public"    <---将所有来自/public 的文件作为静态文件
  }
]
}
},
"EBServerApplicationVersion": {
  "Type": "AWS::ElasticBeanstalk::ApplicationVersion",
  "Properties": {
    "ApplicationName": {"Ref": "EBServerApplication"},
    "Description": "Imagery server: AWS in Action: chapter 13",
    "SourceBundle": {
      "S3Bucket": "awsinaction",
      "S3Key": "chapter13/build/server.zip"    <---从本书的S3 桶中加载代码
    }
  }
},
"EBServerEnvironment": {
  "Type": "AWS::ElasticBeanstalk::Environment",
  "Properties": {
    "ApplicationName": {"Ref": "EBServerApplication"},
    "Description": "Imagery server: AWS in Action: chapter 13",
    "TemplateName": {"Ref": "EBServerConfigurationTemplate"},
    "VersionLabel": {"Ref": "EBServerApplicationVersion"}
  }
}
}

```

引擎下，Elastic Beanstalk使用ELB将流量分发到也由Elastic Beanstalk管理的EC2实例。只需要担心Elastic Beanstalk的配置和代码。



## 4. 用于工作进程的Elastic Beanstalk

工作进程的Elastic Beanstalk应用和服务端很像。其区别将在代码清单13-9中突出显示。

代码清单13-9 Imagery CloudFormation模板：用于用户进程的Elastic Beanstalk

```
"EBWorkerApplication": {      <---描述工作进程应用容器
  "Type": "AWS::ElasticBeanstalk::Application",
  "Properties": {
    "ApplicationName": "imagery-worker",
    "Description": "Imagery worker: AWS in Action: chapter 13"
  }
},
"EBWorkerConfigurationTemplate": {
  "Type": "AWS::ElasticBeanstalk::ConfigurationTemplate",
  "Properties": {
    "ApplicationName": {"Ref": "EBWorkerApplication"},
    "Description": "Imagery worker: AWS in Action: chapter 13",
    "SolutionStackName":
      "64bit Amazon Linux 2015.03 v1.4.6 running Node.js",
    "OptionSettings": [{
      "Namespace": "aws:autoscaling:launchconfiguration",
      "OptionName": "EC2KeyName",
      "Value": {"Ref": "KeyName"}
    }, {
      "Namespace": "aws:autoscaling:launchconfiguration",
      "OptionName": "IamInstanceProfile",
      "Value": {"Ref": "WorkerInstanceProfile"}
    }, {
      "Namespace": "aws:elasticbeanstalk:sqs",
      "OptionName": "WorkerQueueURL",
      "Value": {"Ref": "SQSQueue"}
    }, {
      "Namespace": "aws:elasticbeanstalk:sqs",
      "OptionName": "HttpPath",
      "Value": "/sqs"      <---配置HTTP 资源，在SQS 消息收到时调用
    }, {
      "Namespace": "aws:elasticbeanstalk:container:nodejs",
      "OptionName": "NodeCommand",
      "Value": "node worker.js"
    }, {
      "Namespace": "aws:elasticbeanstalk:application:environment",
      "OptionName": "ImageQueue",
      "Value": {"Ref": "SQSQueue"}
    }
  ]
}
```

```

    }, {
      "Namespace": "aws:elasticbeanstalk:application:environment",
      "OptionName": "ImageBucket",
      "Value": {"Ref": "Bucket"}
    }
  ]
},
"EBWorkerApplicationVersion": {
  "Type": "AWS::ElasticBeanstalk::ApplicationVersion",
  "Properties": {
    "ApplicationName": {"Ref": "EBWorkerApplication"},
    "Description": "Imagery worker: AWS in Action: chapter 13",
    "SourceBundle": {
      "S3Bucket": "awsinaction",
      "S3Key": "chapter13/build/worker.zip"
    }
  }
},
"EBWorkerEnvironment": {
  "Type": "AWS::ElasticBeanstalk::Environment",
  "Properties": {
    "ApplicationName": {"Ref": "EBWorkerApplication"},
    "Description": "Imagery worker: AWS in Action: chapter 13",
    "TemplateName": {"Ref": "EBWorkerConfigurationTemplate"},
    "VersionLabel": {"Ref": "EBWorkerApplicationVersion"},
    "Tier": {      <--- 切换到工作进程环境层（将SQS 消息推送到应用程序）
      "Type": "SQS/HTTP",
      "Name": "Worker",
      "Version": "1.0"
    }
  }
}
}

```

在所有的JSON读取之后，应该创建CloudFormation堆栈。验证堆栈的状态如下：

```

$ aws cloudformation describe-stacks --stack-name imagery
{
  "Stacks": [{
    [...]
    "Description": "AWS in Action: chapter 13",
    "Outputs": [{
      "Description": "Load Balancer URL",
      "OutputKey": "EndpointURL",

```

```
        "OutputValue": "awseb-...582.us-east-1.elb.amazonaws.com"    <-- 将输出复制到浏览器中
    }],
    "StackName": "imagery",
    "StackStatus": "CREATE_COMPLETE"    <-- 等到变为CREATE_COMPLETE
  ]]
}
```

堆栈的**EndpointURL** 输出是用于访问**Imagery**应用程序的URL地址。当在浏览器中打开**Imagery**，可以上传图片，如图13-16所示。

继续上传一些图片。你已创建了一个容错的应用程序！

#### 资源清理

找到12位数字的账号ID（878533158213），可以使用下面的命令行：

```
$ aws iam get-user --query "User.Arn" --output text
arn:aws:iam::878533158213:user/mycli
```

通过执行删除S3桶中所有文件**s3://imagery-\$AccountId**（用账号ID替代**\$AccountId**）：

```
$ aws s3 rm s3://imagery-$AccountId --recursive
```

执行以下命令删除CloudFormation堆栈：

```
$ aws cloudformation delete-stack --stack-name imagery
```

堆栈删除将会花一点儿时间。

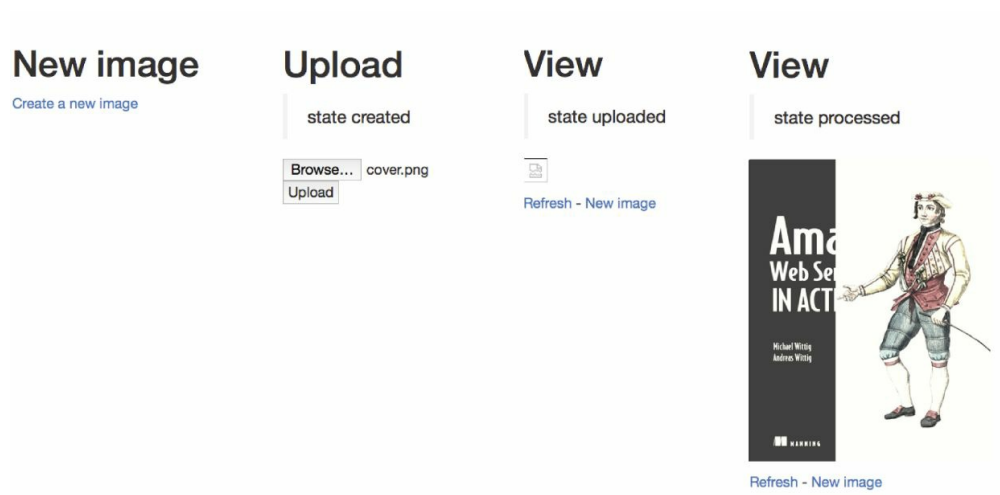


图13-16 Imagery应用程序执行

## 13.4 小结

- 容错意味着假设故障发生。通过这种方式设计系统可以处理故障。
- 要创建容错应用程序，可以使用幂等操作从一个状态转换到下一个状态。
- 状态不应驻留在服务器（无状态服务器）上，这作为容错的先决条件。
- AWS提供容错服务，并提供创建容错系统所需的所有工具。EC2是少数几个不是容错的服务之一。
- 可以使用多个EC2实例消除单点故障。不同可用区域中的冗余EC2实例（以自动扩展组开始）是使EC2容错的方法。

## 第14章 向上或向下扩展：自动扩展和CloudWatch

### 本章主要内容

- 通过启动配置创建一个自动扩展组
- 通过自动扩展调整虚拟服务器的数量
- 在ELB后面扩展同步解耦应用程序
- 利用SQS服务扩展异步解耦的应用
- 利用CloudWatch的告警修改自动扩展组

假设你要组织一个生日晚会，你要购买多少饮料和食物？准确地预测购物的数量是很困难的。

- 有多少客人会参加？有些客人已经确定要来，但是有些人会在晚会前取消出席，甚至有些人还不会提前告知你。因此实际出席客人的数量是不确定的。
- 你的客人会吃多少食物，喝多少饮料？那天会是一个大热天，大家喝很多吗？你的客人会饿着吗？你只能根据以往晚会的经验猜测食物和饮料的量。

由于很多未知的因素，解决这种预算问题确实很难。为了给大家留下一个好的印象，你会预先购买比实际需求要多的食物和饮料，希望大家一个美好的自助生日餐，这样就不会有人饿着或者渴着回家。

满足未来需求的计划几乎是不可能的。为了避免供需之间的差距，你将会在峰值需求上再加上额外的数量来避免未来的资源短缺。

当我们规划IT设施的容量时，我们也会做出一样的行为。我们购买数据中心的硬件设施时，经常是基于未来的需求来进行购买的。

然而，当我们做决定时，我们同样会遇到很多不确定的因素。

- IT基础设施需要服务多少用户？

- 用户会需要多少存储空间？
- 为满足用户的要求，需要多大的计算能力？

为了避免供求之间的差距，不得不订购更多、更快的硬件设施来增加不是非常必要的开销。

在AWS上，你可以按需使用云计算的服务，因此提前预测容量已经没那么重要了，服务器从一个扩容到成千个服务器是完全可行的。存储容量可以自动从GB级别扩容到PB级别，可以按需扩容，因此也不需要容量预估。这种按需扩容的特性就是我们强调的“弹性”。

像AWS一样的公有云厂商短时间内可以提供你所需的容量。AWS已经服务于一百多万的客户，在此规模下，分钟级别之内给你同时提供100个虚拟服务器是完全可能的。这就解决了典型的流量模式问题，如图14-1所示。假想一下白天与晚上你的基础设施上负载的容量区别；周末与平时的区别；圣诞前与其他时间的区别。如果当你在流量上升时能够增加容量，流量下降的时候减小容量，这岂不是更好的做法。在本章中，你将学会基于现在的负载如何调节虚拟服务器。

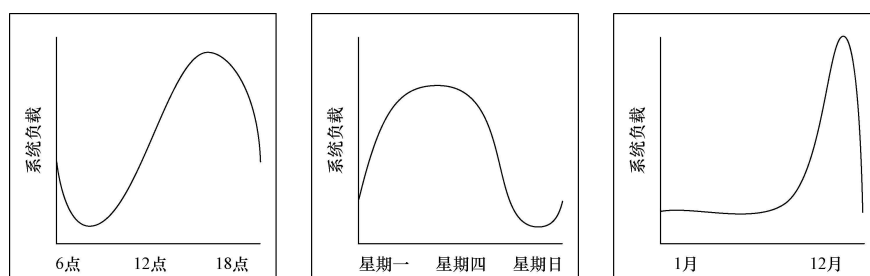


图14-1 网上商店的典型流量模式

扩展虚拟服务器数量可以在AWS上通过自动扩展（auto-scaling）和扩展策略（scaling policy）来实现。自动扩展是EC2服务的一部分，可满足扩展系统当前负载所需的EC2实例的数量的需要。我们在第11章中介绍过自动扩展能够在即使整个数据中心中断的情况下，也能够保证单个虚拟服务器的正常运行。在本章中，读者将了解如何使用动态服务器池。

- 使用自动扩展启动多个相同类型的虚拟服务器。
- 借助CloudWatch的功能，依据CPU负载更改虚拟服务器数量。
- 基于计划更改虚拟服务器的数量，以便能够适应循环的流量模式。

- 使用负载均衡器作为动态服务器池的入口点。
- 使用队列把从动态服务器池来的任务解耦。

#### 示例都包含在免费套餐中

本章中的所有示例都包含在免费套餐中。只要不是运行这些示例好几天，就不需要支付任何费用。记住，这仅适用于读者为学习本书刚刚创建的全新AWS账户，并且在这个AWS账户里没有其他活动。尽量在几天的时间里完成本章中的示例，在每个示例完成后务必清理账户。

这里有两个先决条件使得能够水平扩展应用程序，这就意味着增加和减少虚拟服务器数量是基于如下工作负载。

- 要扩展的服务器需要是无状态的。可以把数据存储在RDS（SQL数据库），DynamoDB（NoSQL数据库）或S3（对象存储）之类的服务，而不是存储在只有特定的服务器可以识别的本地或者网络连接的磁盘上，这样可以使服务器处于无状态。
- 需要动态服务器池的入口点能够在多个服务器上分配流量。服务器可以与负载均衡器同步解耦或与队列异步解耦。

我们在本书的第三部分中介绍了无状态服务器的概念，并在第12章中解释了如何使用解耦。本章会重新学习无状态服务器的概念，并通过一个实际例子了解同步和异步解耦。



## 14.1 管理动态服务器池

设想一下，需要提供可扩展的基础架构来运行像博客平台一样的Web应用程序。当请求数量增加时，需要启动环境一致的虚拟服务器，而在请求数量减少时，终止闲置的虚拟服务器。为了自动化适应当前流量负载，需要能够自动启动和终止虚拟服务器。博客平台的配置和部署需要在启动配置期间完成，无须人工干预。

AWS提供了自动扩展来管理这种动态服务器池。自动扩展可帮你实现以下功能

- 运行可以动态调整所需数量的虚拟服务器。
- 启动、配置和部署统一环境的虚拟服务器。

如图14-2所示，自动扩展组包括以下3个部分。

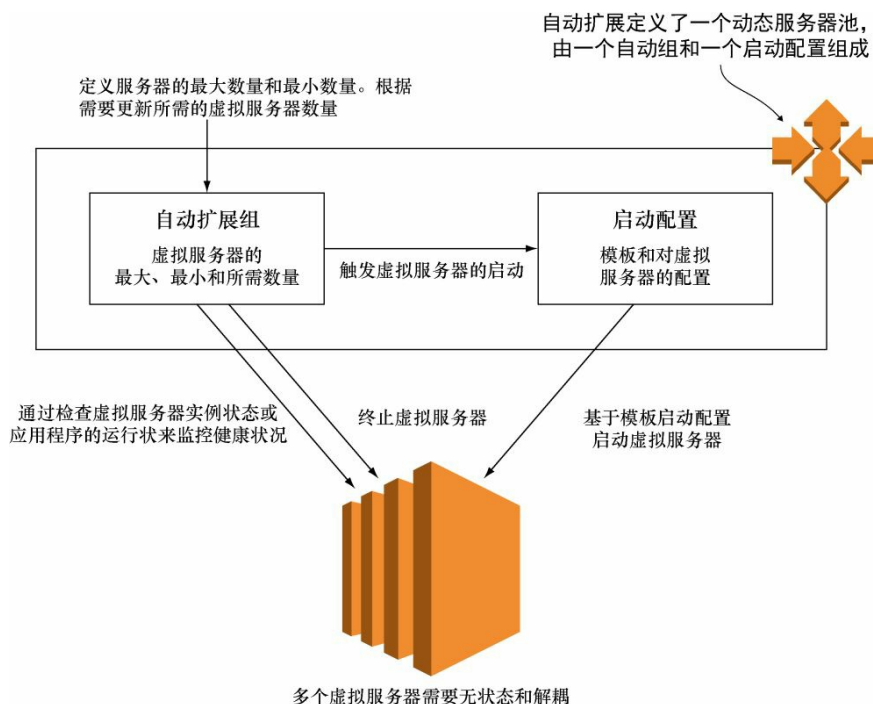


图14-2 自动扩展包括自动扩展组和启动配置，启动和终止统一虚拟服务器

- 确定虚拟服务器的大小，映像和配置的启动配置。

- 确定基于启动配置需要运行多少个虚拟服务器的自动扩展组。
- 调整在自动扩展组里所需服务器数量的扩展策略。

由于自动扩展需要参考启动配置，因此在创建自动扩展组之前需要先创建启动配置。如果使用模板（像本章一样），启动配置直接由CloudFormation自动解决。

如果希望用很多服务器来解决工作负载，则必须启动相同的虚拟服务器来构建同质的基础设施。可以使用启动配置来定义和配置新的虚拟服务器。表14-1显示了启动配置的最重要的参数。

表14-1 启动配置参数

名 称	描 述	可 能 值
ImageIdID	启动一个新的虚拟服务器所需要的映像	Amazon系统映像（AMI）的ID
InstanceType	新虚拟服务器的大小	虚拟服务器类型（如t2.micro）
UserData	在引导期间执行脚本的虚拟服务器的用户数据	Base64编码字符串
KeyName	密钥对的名称	密钥对的名称
AssociatePublicIpAddress	公网IP关联到虚拟服务器上	真或者假
SecurityGroups	安全组关联到新的虚拟服务器上	安全组名称的清单
IamInstanceProfile	IAM实例文件关联到IAM角色	名称或者IAM实例性能的ARN
SpotPrice	以最大价格使用竞价实例，而不用按需示例	竞价实例每小时最高价（如0.1）

Ebs Optimized	通过AMI设定，给EBS根卷提供专用吞吐量来使EC2能够EBS优化	真或者假
---------------	-----------------------------------	------

创建启动配置后，可以创建自动扩展组来启动配置。自动扩展组定义了虚拟服务器的最大、最小值和所需数量。“所需”的意思是应该运行的虚拟服务器的数量。如果当前服务器数量低于所需数量，则自动扩展组将添加服务器。如果当前服务器数量高于所需数量，服务器将被终止。

自动扩展组还会监视EC2实例是否正常工作，并替换损坏的实例。表14-2显示了自动扩展组一些重要的参数。

表14-2 自动扩展组重要参数

名 称	描 述	可 能 值
DesiredCapacity	所需健康的虚拟服务器	整数
MaxSize	虚拟服务器的最大数量，扩展上限	整数
MinSize	虚拟服务器的最小数量，扩展下限	整数
Cooldown	扩大与缩小之间的最小时间跨度	秒数
HealthCheckType	自动扩展组如何监测虚拟服务的健康状况	EC2（实例的健康）或者ELB（由负载均衡器完成的健康检查）
HealthCheckGracePeriod	新启动一个实例之后停止健康监测到引导启动配置的时间间隔	秒数

LaunchConfigurationName	用来启动新的虚拟服务器的启动配置名称	启动配置的名称
LoadBalancerNames	负载均衡器自动注册新实例	负载均衡器名列表
TerminationPolicies	用来确定哪一个实例先终止的策略	OldestInstance、NewestInstance、OldestLaunchConfiguration、Closest To Next Instance Hour或Default
VPCZoneIdentifier	启动EC2实例的子网列表	VPC的子网列表

如果在**VPCZoneIdentifier**的帮助下为自动扩展组指定多个子网，EC2实例将均匀分布在这些子网之间，从而在可用区之间均匀分布。

#### 避免不必要的**cooldown**和宽限期扩展

请务必定义合理的**Cooldown**和**HealthCheckGracePeriod**值。建议指定较短的**Cooldown**和**HealthCheckGracePeriod**周期。但是如果你的**Cooldown**时间太短，你会过早地被放大和缩小。如果你的**HealthCheckGracePeriod**太短，自动扩展组将启动一个新实例，因为上一个实例不能快速启动。两者都将启动不必要的实例并导致不必要的费用。

通常，你无法编辑启动配置。如果需要更改启动配置，请按照下列步骤操作。

- (1) 创建新的启动配置。
- (2) 编辑自动扩展组，并引用新的启动配置。
- (3) 删除旧的启动配置。

幸运的是，当你对模板中的启动配置进行更改时，CloudFormation会为你执行此操作。代码清单14-1展示了如何在CloudFormation模板的帮助下设置此类动态服务器池。

```
[...]
"LaunchConfiguration": {
  "Type": "AWS::AutoScaling::LaunchConfiguration",
  "Properties": {
    "ImageId": "ami-b43503a9",      <--操作系统映像（AMI）启动新的虚拟服务器
    "InstanceType": "t2.micro",      <--新的EC2 实例的实例类型
    "SecurityGroups": ["webapp"],    <--用于新的虚拟服务器的密钥对的名称
    "KeyName": "mykey",             <--启动虚拟服务器附加这些安全组
    "AssociatePublicIpAddress": true, <--用新的虚拟服务器关联一个公有IP
地址
    "UserData": {"Fn::Base64": {"Fn::Join": ["", [      <--虚拟服务器引导期间
执行的脚本
      "#!/bin/bash -ex\n",
      "yum install httpd\n",
    ]}}}
  }
},
"AutoScalingGroup": {
  "Type": "AWS::AutoScaling::AutoScalingGroup",
  "Properties": {
    "LoadBalancerNames": [{"Ref": "LoadBalancer"}],    <--在ELB 注册新的虚
拟服务器
    "LaunchConfigurationName": {"Ref": "LaunchConfiguration"},    <--引用
启动配置
    "MinSize": "2",      <--服务器的最小数量
    "MaxSize": "4",      <--服务器的最大数量
    "DesiredCapacity": "2",    <--自动扩展组试图达到的健康的虚拟服务器数量
    "Cooldown": "60",      <--在两个扩展操作之间等待60 s（如启动新的虚拟服务器
）
    "HealthCheckGracePeriod": "120",    <--在虚拟服务器启动后等待120 s，然后
开始监视其运行状况
    "HealthCheckType": "ELB",    <--使用ELB的健康检查EC2 实例的运行状况
    "VPCZoneIdentifier": ["subnet-a55fafcc", "subnet-fa224c5a"]    <--在V
PC 的这两个子网中启动虚拟服务器
  }
}
[...]
```

如果需要在多个可用区域中启动同一类型的多个虚拟服务器，则自动扩展组是一个非常好用的工具。

## 14.2 使用监控指标和时间计划触发扩展

到目前为止，在本章中，你已经学习了如何使用自动扩展组和启动配置来启动虚拟服务器。你可以手动更改自动扩展组的所需容量，并且将启动新实例或终止旧实例以达到新的所需容量。

要为博客平台提供可扩展的基础架构，你需要通过使用扩展策略调整自动扩展组的所需容量，因而自动增加和减少动态服务器池中的虚拟服务器数量。

许多人在午休期间进行网上冲浪，因此你可能需要在每天的上午11:00和下午1:00之间增加虚拟服务器。你还需要适应不可预测的负载模式，例如，在你的博客平台上发表的文章在社交媒体上得到广泛传播，在架构上你需要做出调整。

图14-3说明了更改虚拟服务器数量的两种不同方法。

- 根据度量参数（如CPU利用率或负载均衡器上的请求数）使用CloudWatch警报增加或减少虚拟服务器数量。
- 根据重复的负载模式设定计划增加或减少虚拟服务器的数量（如夜间减少虚拟服务器的数量）。

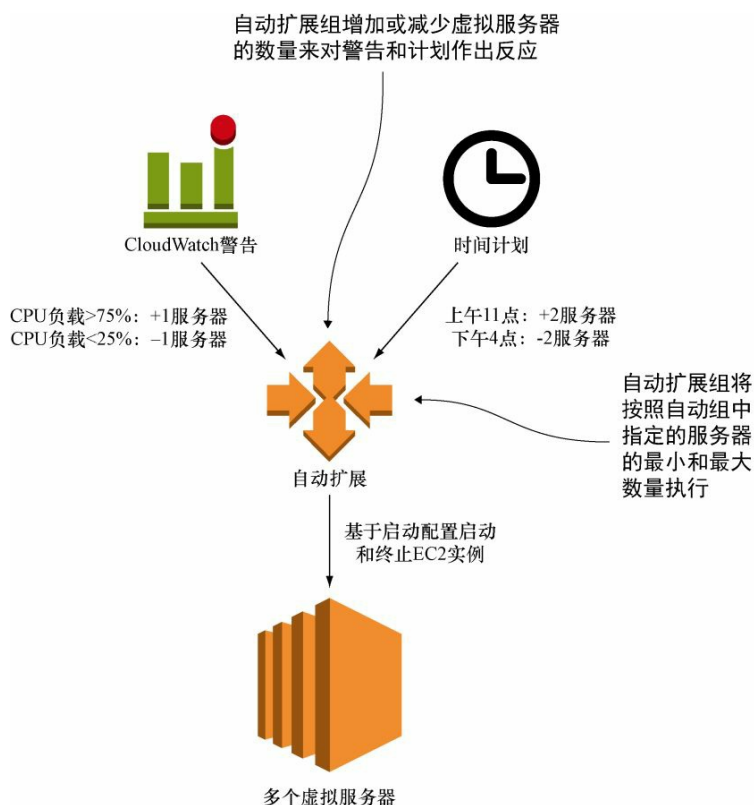


图14-3 基于CloudWatch警报或计划触发自动多个虚拟服务器

基于时间计划的扩展比CloudWatch指标的扩展更为简单，因为CloudWatch里很难找到准确的扩展指标。然而基于时间计划的扩展也不太精确。

## 14.2.1 基于时间计划的扩展

管理博客平台时，你可能会注意到重复的负载模式。

- 许多人似乎在午休时间（上午11:00至下午1:00之间）阅读文章。
- 当你在晚上投放电视广告后，你的注册页面的请求会大幅增加。

你可以使用不同类型的时间计划扩展操作来对系统使用中的模式做出反应。

- 一次性操作，通过**Starttime** 参数来创建。
- 循环操作，通过**recurrence** 参数来创建。

可以在命令行的帮助下创建两种类型的时间计划扩展操作。代码清单14-2中显示的命令行创建了一个时间计划扩展操作，具体内容是在2016年1月1日12:00（UTC）将自动扩展组（名为**webapp**）所需的容量设置为**4**。不要立即运行如下命令，因为你尚未创建自动扩展组的Web应用程序。

代码清单14-2 计划一次性的扩展操作

```
$ aws autoscaling put-scheduled-update-group-action \  
--scheduled-action-name ScaleTo4 \  
--auto-scaling-group-name webapp \  
--start-time "2016-01-01T12:00:00Z" \  
--desired-capacity 4
```

←-- 计划扩展操作的名称  
←-- 自动扩展组的名称  
←-- 出发扩展操作的启动时间（UTC）  
←-- 需要为计划扩展组设置容量

你还可以使用cron语法执行时间计划循环扩展操作。通过命令行将自动扩展组所需的容量设置为每天20:00（UTC）为2，如代码清单14-3所示。不要立即运行如下命令，因为你尚未创建自动扩展组**webapp**。

代码清单14-3 计划每天在UTC时间20:00点运行的定期扩展操作

```
$ aws autoscaling put-scheduled-update-group-action \  
--scheduled-action-name ScaleTo2 \  
--auto-scaling-group-name webapp \  
--recurrence "0 20 * * *" \  
--desired-capacity 2
```

←-- 计划扩展操作的名称  
←-- 自动扩展组的名称  
←-- 按Unix cron 语法中的定义那样，每天在20:00（UTC）触发一个动作  
←-- 要为计划扩展组设置容量

循环是在Unix cron语法格式中定义的，如下所示：

```
* * * * *  
| | | | |  
| | | | +- day of week (0 - 6) (0 Sunday)  
| | | +--- month (1 - 12)  
| | +----- day of month (1 - 31)  
| +----- hour (0 - 23)  
+----- min (0 - 59)
```



你也可以添加另一个时间计划的循环扩展操作，早晨增加而晚上减少容量。只要在某一段时间内你的基础架构上的负载可预测，你就可以使用时间计划来进行扩展操作。例如，内部系统可能在工作时间内容量最大，或者市场营销活动在既定时间内上线。

## 14.2.2 基于CloudWatch参数的扩展

预测未来是一项艰巨的任务。流量超出我们已知模式的时候，流量会不时地增加或减少。例如，如果你博客平台上发布文章得到各大社交媒体的转载，你需要对这预期之外的负载做出反应，并迅速扩展服务器数量。

你可以借助CloudWatch和扩展策略来调整EC2实例的数量来处理当前的工作负载，如图14-4所示。CloudWatch帮助你监控AWS上的虚拟服务器和其他服务的性能。通常，各个服务将使用量的参数发送到CloudWatch上，以便帮你评估可用容量。根据当前的工作负载触发扩展，而你需要使用的是参数、警报和扩展策略。

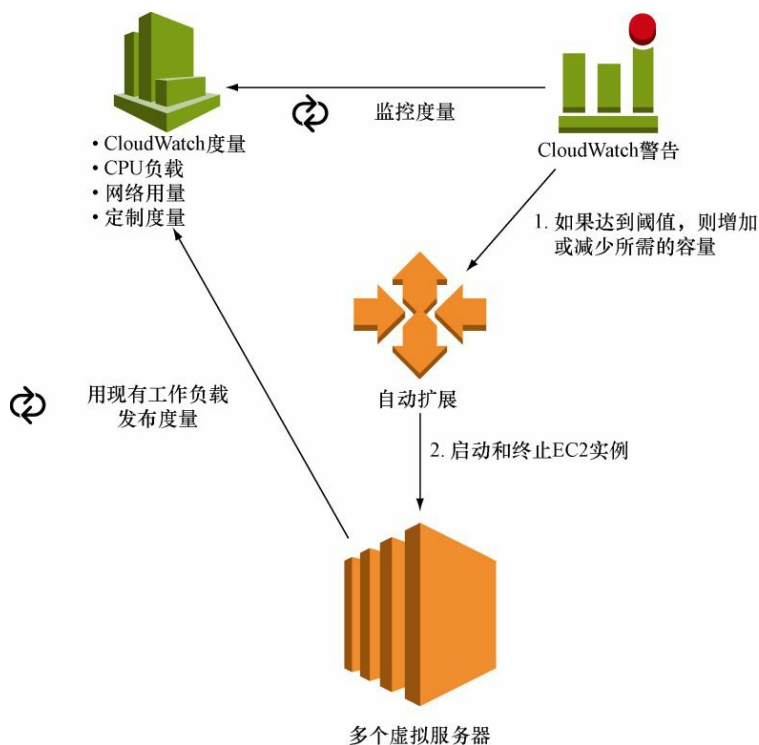


图14-4 基于CloudWatch的参数和报警的触发自动扩展

EC2实例默认向CloudWatch发送几个重要指标，像CPU、网络 and 磁盘利用率。然而，目前还没有虚拟服务器内存使用的相关指标信息。如果达到瓶颈，你可以使用上述3个指标来扩展服务器数量。例如，如果CPU利用率达到极限，则添加服务器。

下列参数描述了CloudWatch指标。

- **Namespace** ——指定指标的来源（如AWS/EC2）。
- **Dimensions** ——指定指标范围（如所有自动扩展组里的虚拟服务器）。
- **MetricName** ——指标唯一的名称（如CPUUtilization）。

一个CloudWatch警报是基于一个CloudWatch指标的。表14-3详解了警报的相关参数。

表14-3 CloudWatch基于CPU的利用率触发自动扩展组里所有服务器的扩容的警报参数

上下文	名称	描述	可能值
条件	statistic	度量的统计函数应用	Average、Sum、Minimum-Maximum、SampleCount
	Period	从度量中定义基于时间的值切片	s（倍数为60）
条件	EvaluationPeriods	检查警报时要评估的期间数	整数
	Threshold	警告阈值	数值
	ComparisonOperator	运算符将阈值与统计函数的结果进行比较	GreaterThanEqualToThreshold、GreaterThanOrEqualToThreshold、LessThanThreshold、LessThanOrEqualToThreshold
	Namespace	度量的来源	AWS/EC2 EC2服务的指标

指标	Dimensions	度量的范围	取决于度量，参照聚合度量的自动扩展自动扩展组的所有关联的服务器
	MetricName	度量的名称	例如：CPUUtilization
动作	AlarmActions	如果达到阈值触发行为	扩展策略的引用

代码清单14-4创建一个警报，如果所有自动扩展组里的虚拟服务器的平均CPU利用率超过80%，则增加虚拟服务器的数量。

代码清单14-4 基于自动扩展组CPU负载的CloudWatch报警

```

"CPUHighAlarm": {
  "Type": "AWS::CloudWatch::Alarm",
  "Properties": {
    "EvaluationPeriods": "1",      <--仅计算一个周期
    "Statistic": "Average",      <--计算度量值的平均值
    "Threshold": "80",          <--阈值为80% CPU 利用率
    "AlarmDescription": "Alarm if CPU load is high.",      <--警告的描述
    "Period": "60",            <--一个周期60 s
    "AlarmActions": [{"Ref": "ScalingUpPolicy"}],      <--如果达到阈值，则触
发扩展策略
    "Namespace": "AWS/EC2",      <--度量由EC2 实例发布
    "Dimensions": [{            <--使用从属于特定自动扩展组的所有服务器的CPU 利用率的
度量
      "Name": "AutoScalingGroupName",
      "Value": {"Ref": "AutoScalingGroup"}
    ]},
    "ComparisonOperator": "GreaterThanThreshold",      <--如果CPU 的平均利用
率比阈值高，则触发警报
    "MetricName": "CPUUtilization"      <--包含EC2 实例的CPU利用率的度量
  }
}

```

如果达到阈值，CloudWatch警报将触发一个操作。要将警报与自动扩展组连接，你需要设定扩展策略。扩展策略定义由CloudWatch警报执行的扩展操作。

如图14-5所示，通过CloudFormation创建了一个扩展策略。扩展策略绑定到自动扩展组。下面有3个不同的选项来调整自动扩展组的所需容量。

- **ChangeInCapacity** ——以绝对数值增加或减少服务器数量。
- **PercentChangeInCapacity** ——以百分比增加或减少服务器数量。
- **ExactCapacity** ——将所需容量设置为指定的数量。

代码清单14-5 将在触发时添加一台服务器的扩展策略

```
"ScalingUpPolicy": {
  "Type": "AWS::AutoScaling::ScalingPolicy",
  "Properties": {
    "AdjustmentType": "ChangeInCapacity",      <--按绝对数量更改容量
    "AutoScalingGroupName": {"Ref": "AutoScalingGroup"},  <--引用自动扩
展组
    "Cooldown": "60",      <--等待至少60 s，直到下一个扩展操作可以发生
    "ScalingAdjustment": "1"  <--自动扩展组的所需容量增加1
  }
}
```

你可以在许多不同的指标上定义警报。你将会看到AWS在官方网站提供的所有命名空间、维度和度量的概述。你还可以发布自定义度量标准，例如，直接从应用程序（如线程池使用、处理时间或用户会话）中的度量。

#### 基于虚拟服务器CPU负载扩展，提供突发性能

一些像t2通用系列的实例可以提供突发性能。这些虚拟服务器在基准CPU性能下提供服务，并且可以拥有基于信用量在短时间内突发的性能。如果所有的积分都被用光了，实例在基准水平工作。对于t2.micro实例，基准性能是底层物理机CPU性能的10%。

使用具有突发性能的虚拟服务器可以帮助你应对负载峰值。你在低负载时可以节省积分，并在高负载时使用积分来提高突发性能。但是，基于CPU负载扩展具有突发性能的虚拟服务器数量是不一定行得通的，因为你的扩展策略必须考虑你的实例是否具有足够的积分以满足突发性能的需要。因此可以考虑基于另一个指标扩展（如请求数量）或使用无突发性能的实例类型。

很多时候，我们更希望扩容速度比缩小速度快一些。我们会考虑每

5 min不止增加一个服务器而增加两个，但每10 min只减少一个服务器。此外，可以通过模拟真实流量测试你的扩展策略。例如，设置访问日志的速度与服务器处理请求的速度一样快。但请记住，服务器需要一些时间才能启动，不要期望自动扩展组可以在几秒钟内使你的容量增加一倍。

你已经学习了如何使用自动扩展来使虚拟服务器数量适应工作负载。下面通过一些实践练习来更好地理解它。

## 14.3 解耦动态服务器池

如果你需要根据需求扩展运行博客平台的虚拟服务器数量，自动扩展组可以帮助你提供所需数量的、统一环境的虚拟服务器。而扩展计划或CloudWatch警报将可以帮你自动地增加或减少所需的服务器数量。但是用户如请求如何到达动态服务器池中的服务器来浏览上面托管的文章？HTTP请求应该在哪里路由？

第12章介绍了解耦的概念：在ELB的帮助下进行同步解耦和在SQS的帮助下的异步解耦。通过解耦可以将请求或消息路由到一个或多个服务器。在动态服务器池中不再可能向单个服务器发送请求。如果要使用自动扩展来增加和减少虚拟服务器的数量，你需要解耦服务器，因为无论有多少服务器在负载均衡器或消息队列后工作，从系统外部可达的接口都需要保持不变。图14-5显示了基于同步或异步解耦如何构建可扩展系统。

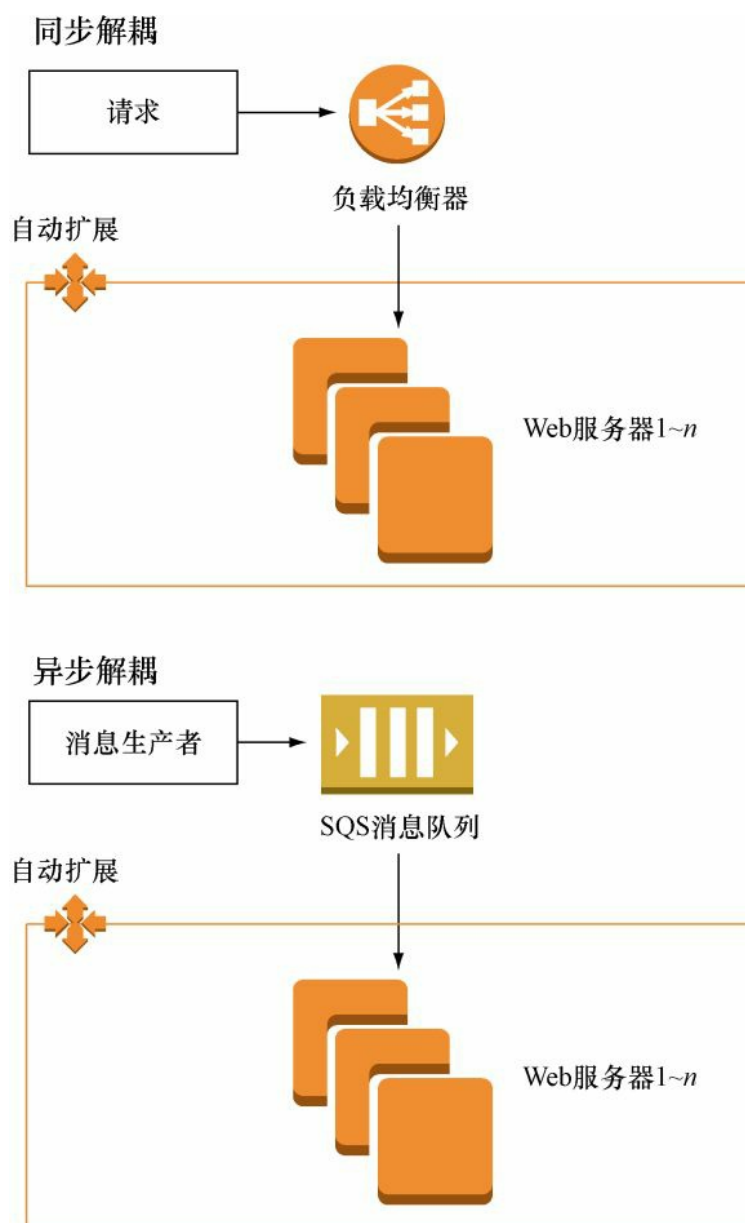


图14-5 解耦允许动态扩展虚拟服务器的数量

可解耦和可扩展应用程序需要无状态的服务器。无状态服务器将在数据库或在存储系统中远程存储数据。以下两个示例解释了无状态服务器的概念。

- WordPress博客 ——与ELB解耦，通过自动扩展组和CloudWatch基于CPU利用率自动扩展，数据保存在外部的RDS和S3中。
- URL2PNG提取URL的屏幕截图 ——与SQS（队列）解耦，通过自动扩展组和CloudWatch基于队列长度自动扩展，数据保存在外部的

DynamoDB和S3。

### 14.3.1 由负载均衡器同步解耦扩展动态服务器池

回应HTTP（S）请求是一个同步任务。如果用户想要使用你的Web应用程序，Web服务器必须立即响应请求。当使用动态服务器池运行Web应用程序时，通常使用负载均衡器将服务器与用户请求解耦。负载均衡器作为动态服务器池的单个入口点，将HTTP（S）请求转发到多个服务器。

假设你的公司正在使用企业博客发布公告并在网络社区上与公众进行互动。你负责博客的托管。晚上流量达到每日的高峰时，营销部门抱怨网页速度很慢。你希望使用AWS的弹性根据当前工作负载扩展服务器数量来解决网速的问题。

你的公司在WordPress上部署企业博客。第2章和第9章介绍了基于EC2实例和RDS（MySQL数据库）的WordPress安装程序。在本书的最后一章中，我们将通过添加扩展的能力来完成这个例子。

图14-6显示了可扩展的WordPress示例。以下服务使用了高度可用的扩展体系架构。

- 运行Apache的EC2实例提供PHP应用程序WordPress。
- RDS提供了一个通过多可用区部署高度可用的MySQL数据库。
- S3存储媒体文件，如图片和视频，与WordPress插件集成。
- ELB同步将Web服务器与访客解耦。
- 自动扩展和CloudWatch基于所有正运行的虚拟服务器的当前CPU负载来扩展Web服务器的数量。



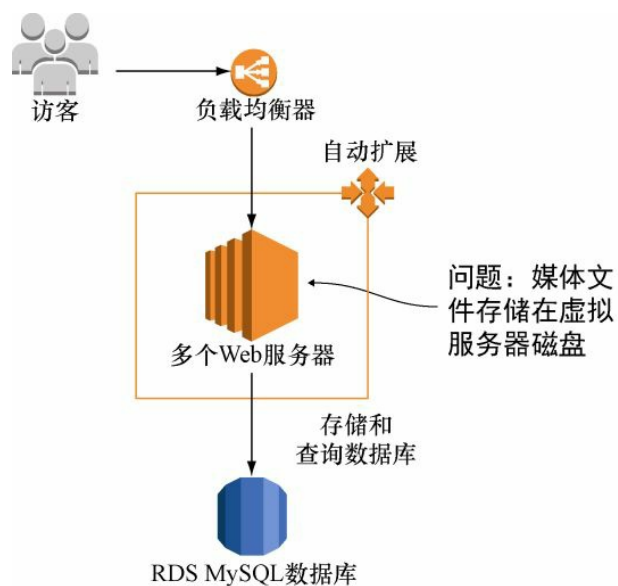


图14-6 在多个虚拟服务器上运行的WordPress、RDS存储数据、虚拟服务器磁盘上存储媒体文件

到目前为止，WordPress示例不能基于当前负载进行扩展，并且还存在一个问题：WordPress将上传的媒体文件存储在本地文件系统中，如图14-7所示。因此，服务器不是无状态的。如果你上传某个博客的图片，则该图片只能在单个服务器上使用。

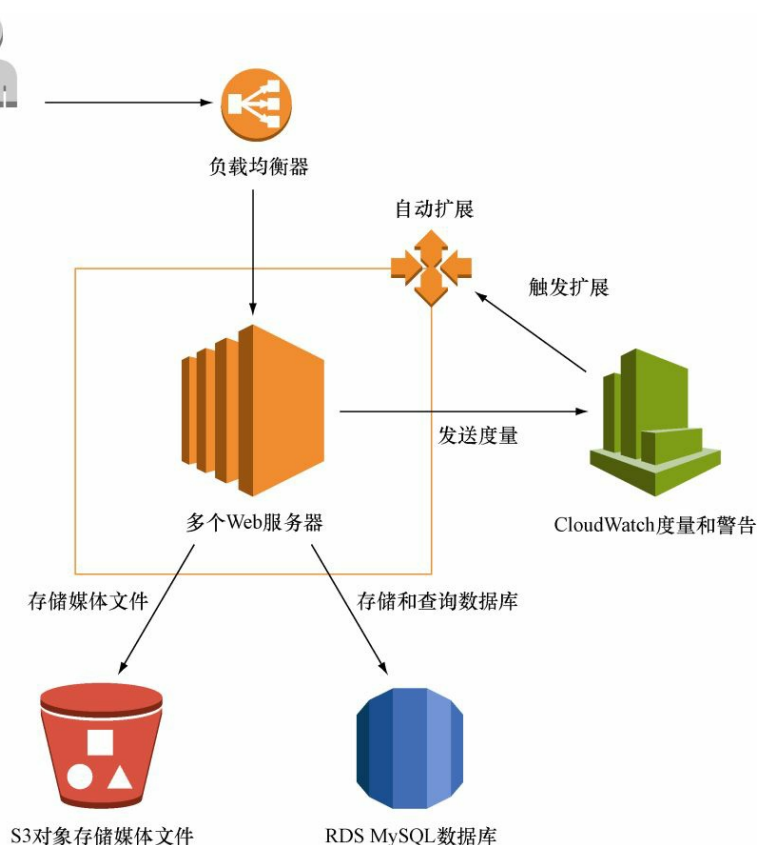


图14-7 自动扩展Web服务器运行的WordPress，数据存储在RDS和S3实现解耦，基于负载均衡器的负载比例上的负荷

如果你想运行多个服务器来处理负载，这将会成为一个问题。其他服务器将无法为上传的图片提供服务，并会显示404（找不到）状态码。要解决这个问题，你将会安装一个名为amazon-s3-cloudfront 的WordPress插件，在S3的帮助下存储和传送媒体文件。服务器的各种服务的状态，就像把数据任务外包给MYSQL的RDS数据库一样，你会把服务器的各种任务外包给其他相应的服务。图14-7显示了WordPress的改进架构版本。

像往常一样，读者可在下载的源代码中找到相关代码。WordPress示例的CloudFormation模板位于/chapter14/wordpress.json中。

执行以下命令创建一个CloudFormation堆栈，用于启动可扩展的WordPress安装程序。使用你的博客的唯一ID（如awsinaction-andreas），`$AdminPassword`（使用随机密码）和`$AdminEMail`（使用你的电子邮件地址）替换`$ BlogID`：

```
$ aws cloudformation create-stack --stack-name wordpress \
--template-url https://s3.amazonaws.com/\
awsinaction/chapter14/wordpress.json \
--parameters ParameterKey=BlogID,ParameterValue=$BlogID \
ParameterKey=AdminPassword,ParameterValue=$AdminPassword \
ParameterKey=AdminEmail,ParameterValue=$AdminEmail \
--capabilities CAPABILITY_IAM
```

创建堆栈最多需要10 min。这期间你可以喝点咖啡或者茶水放松一下。登录AWS管理控制台，到AWS CloudFormation服务来观察名为wordpress 的CloudFormation堆栈的创建过程。你可以浏览CloudFormation模板最重要的两个部分，如代码清单14-6和代码清单14-7所示。

代码清单14-6 可伸缩和高可用WordPress设置（第一部分，总共两部分）

```
"LaunchConfiguration": {
  "Type": "AWS::AutoScaling::LaunchConfiguration",      <--创建自动扩展组的启动配置
  "Metadata": [...],
  "Properties": {
    "ImageId": [...]      <--用于启动虚拟服务器的操作系统映像（AMI）
    "InstanceType": "t2.micro",      <--具有虚拟服务器防火墙规则的安全组
    "SecurityGroups": [      <--虚拟服务器的大小
      {"Ref": "WebServerSecurityGroup"}
    ],
    "KeyName": {"Ref": "KeyName"},      <--用于SSH 访问的密钥对
    "AssociatePublicIpAddress": true,      <--将公有IP 地址与虚拟服务器关联
    "UserData": [...]      <--自动安装和配置WordPress 脚本
  }
},
"AutoScalingGroup": {
  "Type": "AWS::AutoScaling::AutoScalingGroup",      <--创建自动扩展组
  "Properties": {
    "LoadBalancerNames": [{"Ref": "LoadBalancer"}],      <--在负载均衡器上注册虚拟服务器
    "LaunchConfigurationName": {      <--引用启动配置
      "Ref": "LaunchConfiguration"
    },
    "MinSize": "2",      <--确保至少有两个虚拟服务器正在运行，一个或两个可用区的高可用性
    "MaxSize": "4",      <--启动不超过4 个虚拟服务器，以节省成本
    "DesiredCapacity": "2",      <--启动两个所需的Web 服务器，如有必要以后由Clo
```

```

udWatch 警报来进行更改
  "Cooldown": "60",          ←--在扩展操作之间至少等待60 s
  "HealthCheckGracePeriod": "120",    ←--在开始监视启动虚拟服务器的运行状
况之前，至少等待120 s
  "HealthCheckType": "ELB",          ←--使用ELB 运行状况检查来监视虚拟服务器的运
行状况
  "VPCZoneIdentifier": [          ←--在两个不同的可用性区中启动虚拟服务器，以获得
高可用性
    {"Ref": "SubnetA"}, {"Ref": "SubnetB"}
  ],
  "Tags": [{          ←--为自动扩展组启动的所有虚拟服务器添加一个包含名称的标记
    "PropagateAtLaunch": true,
    "Value": "wordpress",
    "Key": "Name"
  }]
}
[...]
}

```

扩展策略和CloudWatch警报在代码清单14-7中遵循。

代码清单14-7 可伸缩和高可用的WordPress设置（第二部分，总共两部分）

```

"ScalingUpPolicy": {
  "Type": "AWS::AutoScaling::ScalingPolicy",    ←--创建可由CloudWatch 警报
触发的扩展策略，以增加所需实例的数量
  "Properties": {
    "AdjustmentType": "ChangeInCapacity",    ←--更改所需虚拟服务器的容量
    "AutoScalingGroupName": {          ←--引用自动扩展组
      "Ref": "AutoScalingGroup"
    },
    "Cooldown": "60",          ←--在扩展策略触发的所需容量的两个更改之间至少等待60
s
    "ScalingAdjustment": "1"    ←--将自动扩展组的当前所需容量增加1
  }
},
"CPUHighAlarm": {
  "Type": "AWS::CloudWatch::Alarm",    ←--创建新的CloudWatch 警报以监视CPU
使用情况
  "Properties": {
    "EvaluationPeriods": "1",    ←--平均函数应用于度量
    "Statistic": "Average",    ←--检查警报时要评估的时间
    "Threshold": "60",    ←--定义60% CPU 利用率作为警报的阈值
    "AlarmDescription": "Alarm if CPU load is high.",

```

```

    "Period": "60",          ←--从度量中定义基于时间的60 s 值切片
    "AlarmActions": [{ "Ref": "ScalingUpPolicy" }],      ←--引用扩展策略作为触
发状态更改警报的操作
    "Namespace": "AWS/EC2",      ←--度量的来源
    "Dimensions": [{          ←--度量的范围，在所有关联的服务器上引用自动组进行聚合
度量
        "Name": "AutoScalingGroupName",
        "Value": { "Ref": "AutoScalingGroup" }
    }],
    "ComparisonOperator": "GreaterThanThreshold",      ←--如果平均值大于阈值
，则触发警报
    "MetricName": "CPUUtilization"      ←--使用包含CPU 利用率的度量
    }
},
"ScalingDownPolicy": {      ←--扩展策略向下扩展（相对于扩展策略以扩大规模）
    [...]
},
"CPULowAlarm": {      ←--如果CPU 利用率低于阈值，则CloudWatch 警报
    [...]
}

```

在CloudFormation堆栈达到**CREATE-COMPLETE** 状态后，按照以下步骤操作创建包含图片的新博客帖子。

- （1）选择CloudFormation堆栈wordpress，并切换到Outputs选项。
- （2）使用浏览器打开所显示的URL 链接。
- （3）在搜索框中搜索Login（登录）链接，然后单击它。
- （4）使用用户名admin 和你在使用CLI创建堆栈时指定的密码登录。
- （5）单击左侧菜单中的“帖子”。
- （6）点击“添加”。
- （7）输入标题和文字，然后将图片上传到你的帖子。
- （8）点击发布。

(9) 通过再次输入步骤1中的网址，返回到博客。

现在你可以准备扩容了，我们准备了一个负载测试，将在短时间内向WordPress服务器发送10 000个请求。新启动的虚拟服务器处理这个负载。几分钟后，负载测试完成后，其他虚拟服务器将被关闭。听起来如此好玩，你绝对不能错过。

#### 注意

如果计划进行更大规模的负载测试，可考虑AWS可接受的使用策略，并在开始之前请求获得许可。

#### 简单的HTTP负载测试

我们使用一个名为Apache Bench的工具来执行WordPress安装程序的负载测试。该工具是Amazon Linux软件包存储库中httpd-tools软件包的一部分。

Apache Bench是一个基准测试工具。你可以使用指定数量的线程发送指定数量的HTTP请求。我们使用以下命令进行负载测试，使用两个线程向负载均衡器发送10 000个请求。`$UrlLoadBalancer` 由负载均衡器的URL替换：

```
$ ab -n 10000 -c 2 $UrlLoadBalancer
```

使用以下命令更新CloudFormation堆栈以启动负载测试：

```
$ aws cloudformation update-stack --stack-name wordpress \
--template-url https://s3.amazonaws.com/\
awsinaction/chapter14/wordpress-loadtest.json \
--parameters ParameterKey=BlogID,UsePreviousValue=true \
ParameterKey=AdminPassword,UsePreviousValue=true \
ParameterKey=AdminEmail,UsePreviousValue=true \
--capabilities CAPABILITY_IAM
```

在AWS管理控制台的帮助下，观察以下事情的发生。

(1) 打开CloudWatch服务，然后单击左侧的警报。

(2) 当负载测试开始时，名为wordpress-CPHi Alarm-\* 的报警将在几分钟后到达ALARM 状态。

(3) 打开EC2服务并列出所有EC2实例。注意到另外两个实例的启动。最后，你将会看到5个实例（4个Web服务器和运行负载测试的服务器）。

(4) 回到CloudWatch服务，等待名为wordpress-CPU Alarm-\* 的报警到达ALARM 状态。

(5) 打开EC2服务并列出所有EC2实例。观察到两个额外的实例消失。最后，总共你会看到3个实例（两个Web服务器一个运行负载测试的服务器）。

整个过程需要20 min左右。

你已经看到自动调整的动作：你的WordPress设置可以适应当前的工作负载，也就解决了在晚上缓慢加载页面的问题。

#### 资源清理

执行以下命令删除对应于Wordpress设置的所有资源，记住替换\$BlogID：

```
$ aws s3 rb s3://$BlogID --force
$ aws cloudformation delete-stack --stack-name wordpress
```

## 14.3.2 队列异步解耦扩展动态服务器池

如果你想根据你的工作负载扩展容量，异步解耦动态服务器池提供了一个优势：因为请求不需要立即被响应，你可以将请求放入队列，并根据队列长度扩展服务器数量。这为你提供了一个非常准确的衡量指标。由于它们存储在队列中，加载峰值期间请求也不会丢失。

假设你正在开发一个社交书签服务，用户可以保存和共享其书签。提供预览并且显示链接后面的网站是一个重要的功能。但在晚上，大多

数用户向你的服务添加新书签时，从URL到PNG的转换很慢。客户对预览不会立即显示一定会不太满意。

为了在晚上处理峰值负载，你想要使用自动扩展。为此，你需要对新书签的创建和生成网站预览的过程进行解耦。第12章介绍了一个称为URL2PNG的应用程序，它将URL转换为PNG图片。图14-8显示了该架构，其中包括用于异步解耦的SQS队列和用于存储生成的映像的S3。创建书签将触发以下过程。

- (1) 包含新书签的URL和唯一ID发送到SQS队列消息。
- (2) EC2实例运行Node.js应用程序从SQS队列抓取消息。
- (3) Node.js应用程序加载URL并创建屏幕截图。
- (4) 屏幕截图上传到S3存储桶，对象键设置为唯一ID。
- (5) 用户可以在唯一ID的帮助下直接从S3下载网站的屏幕截图。

CloudWatch警报用于监视SQS队列的长度。如果队列的长度达到5，则启动一个新的虚拟服务器来处理工作负载。如果队列长度小于5，则另一个CloudWatch警报会降低自动扩展组的所需容量。

具体代码可在下载的源代码中找到。URL2PNG示例的CloudFormation模板位于/chapter14/url2png.json中。

执行以下命令创建一个CloudFormation堆栈，用于启动URL2PNG应用程序。将\$ **ApplicationID** 替换为你的应用程序的唯一ID（如url2png-andreas）：

```
$ aws cloudformation create-stack --stack-name url2png \
--template-url https://s3.amazonaws.com/\
awsinaction/chapter14/url2png.json \
--parameters ParameterKey=ApplicationID,ParameterValue=$ApplicationID \
--capabilities CAPABILITY_IAM
```

创建堆栈最多需要5 min。登录AWS管理控制台，然后搜到AWS CloudFormation服务以观察名为url2png 的CloudFormation堆栈的过



程。

CloudFormation模板类似于用于创建同步解耦的WordPress安装程序的模板。代码清单14-8显示了主要区别：CloudWatch警报监视SQS队列的长度，而不是CPU使用情况。

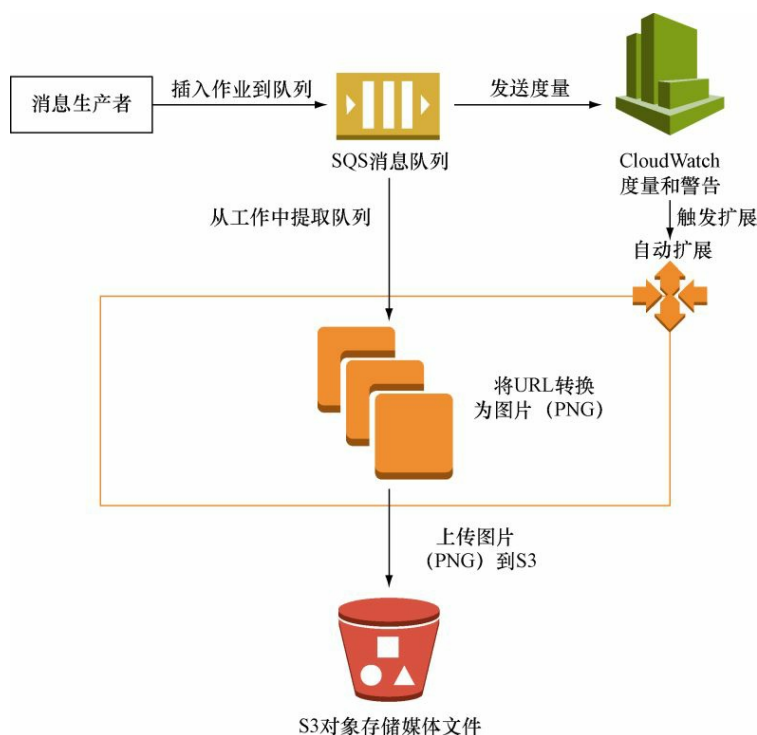


图14-8 将url转换为图片的自动扩展虚拟服务器，由SQS队列解耦

代码清单14-8 监视SQS队列的长度

```
[...]
"HighQueueAlarm": {
  "Type": "AWS::CloudWatch::Alarm",
  "Properties": {
    "EvaluationPeriods": "1",      <--检查警报时要评估的周期数
    "Statistic": "Sum",
    "Threshold": "5",             <--如果达到5 的阈值发出警报
    "AlarmDescription": "Alarm if queue length is higher than 5.",
    "Period": "300",              <--使用300 s 的时间，因为SQS 指标每5 min 发布一次
    "AlarmActions": [{"Ref": "ScalingUpPolicy"}], <--通过扩展策略将所需实例的数量增加1
    "Namespace": "AWS/SQS",       <--该度量的数据由SQS 服务发布
    "Dimensions": [{              <--队列（按名称引用）用作度量的维度
      "Name": "QueueName",
      "Value": {"Fn::GetAtt":
```

```

    ["SQSQueue", "QueueName"]}]
  }],
  "ComparisonOperator": "GreaterThanThreshold",      <-- 如果在该其内值的
总和大于5 这个值，则报警

  "MetricName": "ApproximateNumberOfMessagesVisible"  <-- 度量包含了队列
中挂起的消息的一个大致的数字
}
}
[...]
```

现在你可以进行扩展试验了。我们准备了一个负载测试，将为URL2PNG应用程序快速生成250个消息。将以启动新的虚拟服务器处理负载。几分钟后，负载测试完成后，其他虚拟服务器将会终止。

使用以下命令更新CloudFormation堆栈以启动负载测试：

```

$ aws cloudformation update-stack --stack-name url2png \
--template-url https://s3.amazonaws.com/\
awsinaction/chapter14/url2png-loadtest.json \
--parameters ParameterKey=ApplicationID,UsePreviousValue=true \
--capabilities CAPABILITY_IAM
```

在AWS管理控制台上，观察以下事情的发生。

- (1) 打开CloudWatch服务，然后单击左侧的警报。
- (2) 当负载测试开始时，名为url2png-High Queue Alarm-\* 的警报将在几分钟后到达**ALARM** 状态。
- (3) 打开EC2服务并列出所有EC2实例。注意要启动的其他实例。最后，你将看到三个实例（两个工作服务器和一个运行负载测试的服务器）。
- (4) 回到CloudWatch服务，等待名为url2png-LowQueue Alarm-\* 的警报到达**ALARM** 状态。
- (5) 打开EC2服务并列出所有EC2实例。观察其他实例消失。最

后，你将看到两个实例（一个工作服务器和一个运行负载测试的服务器）。

整个过程需要15 min左右。

你已观察到扩展组的自动调整过程。现在URL2PNG应用程序可以适应当前工作负载，这样也就解决了新书签生成屏幕截图较慢的问题。

#### 资源清理

执行以下命令删除与URL2PNG设置相对应的所有资源（用账号ID替代\$Application）：

```
$ aws s3 rb s3://$ApplicationID --force
$ aws cloudformation delete-stack --stack-name url2png
```

## 14.4 小结

- 可以以相同的方式使用启动配置和自动扩展组来启动多个虚拟服务器。
- EC2、SQS和其他服务将指标参数发送到CloudWatch（CPU利用率、队列长度等）。
- CloudWatch警报可以更改自动扩展组的所需容量。这允许你根据CPU利用率或其他指标增加虚拟服务器的数量。
- 如果要根据当前工作负载扩展服务器，则服务器必须是无状态的。
- 为了在多个虚拟服务器之间分配负载，需要借助于负载均衡器的同步解耦或消息队列的异步解耦。